IoT-SDN-Cloud 참조모델에 기반한 IoT-Cloud 서비스 실증을 위한 SmartX Labs 개발

Document No. 2
Version 0.2
Date 2016-10-08
Author(s) GIST Team

■ 문서의 연혁

버전	날짜	작성자	비고
초안 - 0.2	2016. 10. 08	이준기/mini팀	

본 연구는 한국정보화진흥원(NIA)의 미래네트워크연구시험망(KOREN) 사업 지원과제의 연구결과로 수행되었음 (2016-기술-위20)

This research was one of KOREN projects supported by National Information Society Agency (2016-기술-위20)

Contents

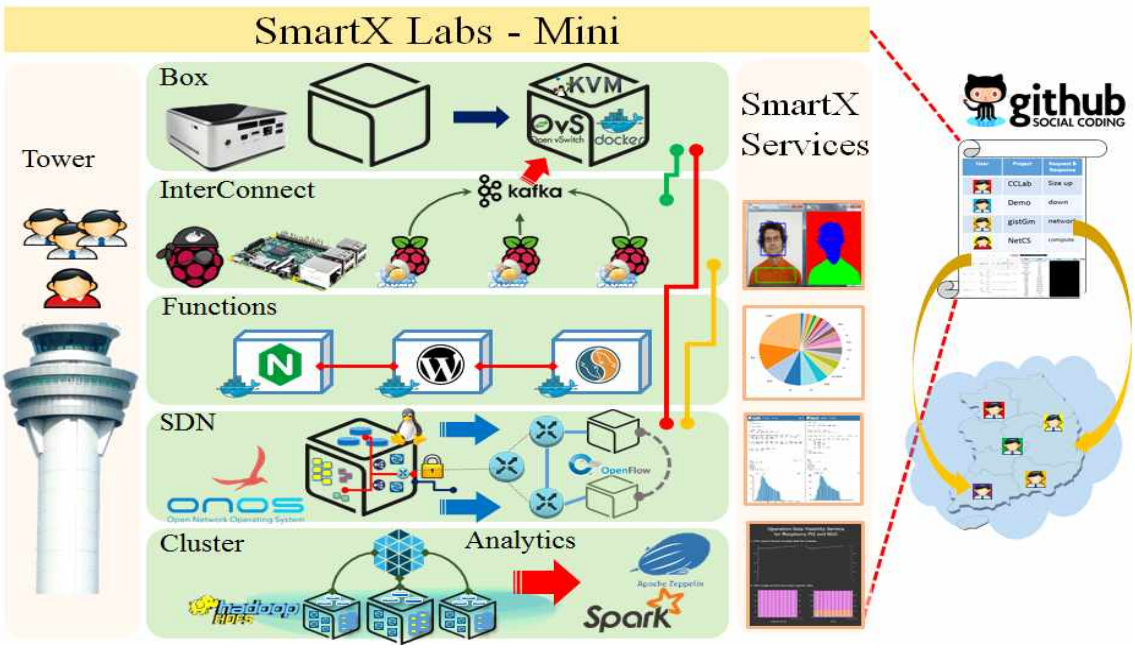
#2. IoT-SDN-Cloud 참조모델에 기반한 IoT-Cloud 서비스 실증을 위한 SmartX Labs 개발

그림 목차

#2. IoT-SDN-Cloud 참조모델에 기반한 IoT-Cloud 서비스 실증을 위한 SmartX Labs 개발

1. SmartX Labs 개요

1.1. 목적 및 개요

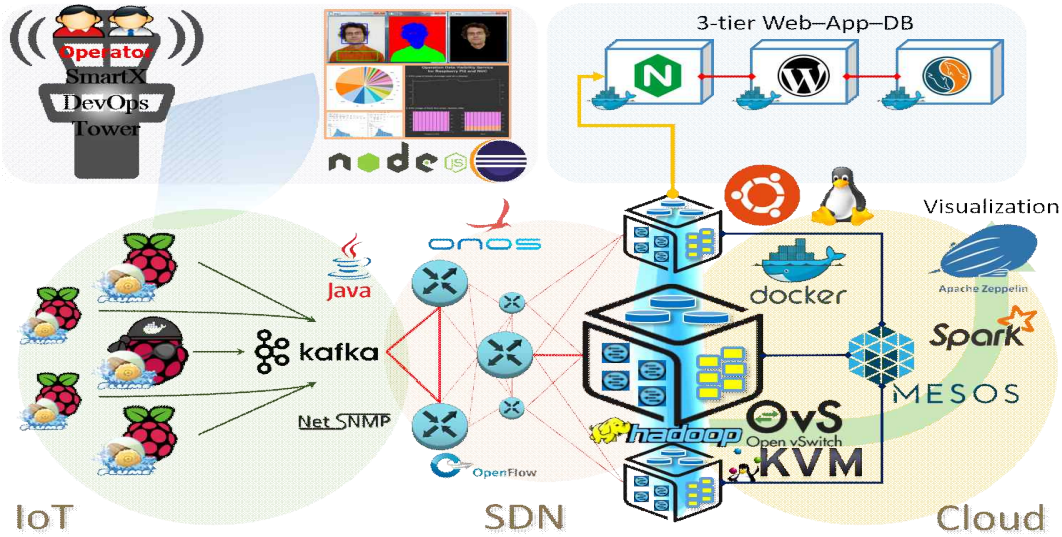


<그림 21: IoT-SDN-Cloud 참조 모델에 기반한 SmartX Labs 구성 및 보급 계획 >

o 본 문서에서는 IoT-SDN-Cloud 참조 모델을 기반으로 KOREN 네트워크 상 OF@KOREN Playground 상에서 실험할 수 있는 자료 SmartX Labs의 개발 방법을 기술한다. 기존 SmartX-mini 모델을 OF@KOREN Playground 상에 공공 IoT-SDN-Cloud 인프라/서비스를 대상으로 상정하여 시험할 수 있도록 SmartX Labs를 개발 및 개선하고, 수행한 시험을 통해 얻어지는 성능, 효율성의 변화를 검증함에 의해 향후 IoT-SDN-Cloud 서비스를 지원하기 위한 참조모델을 도출한다. 또한 SmartX Labs 개발 자료를 기반으로 교육을 진행하며 SmartX Labs 개발 자료를 오픈 포탈 및 GitHub를 통해 외부에 공개하여 보급을 활성화 시킨다.

1.2. SmartX Labs

o 본 기술문서에서 기술하는 SmartX Labs는 IoT-SDN-Cloud 서비스를 구현해보고자 하는 학생들을 타겟으로 . SmartX Labs는 IoT-SDN-Cloud 서비스실증을 위해 각 단계에 따라 IoT, SDN, Cloud 파트로 구성된다. 하드웨어적인 자원으로는 Intel사의 NUV 및 싱글보드 컴퓨터, 그리고 SDN 구축을 위한 Mikrotik 스위치가 사용된다. 또한 IoT-SDN-Cloud 참조모델에 따라 SmartX Labs 시험환경 구축 및 진행을 위해 다양한 오픈소스 프로젝트가 사용된다.



<그림 22: SmartX Labs의 구체적인 구조 및 사용 소프트웨어 >

o SmartX Labs는 Introduction, Box, InterConnect, Tower, Functions, WebApp, Cluster, Analytics, SDN 총 8개의 파트로 나누어져 있으며, GitHub 기반 SmartX Platform에 공개되어 있다. (https://github.com/SmartX-Labs/SmartX-mini)

2. Box Lab

2.1. Box Lab Background

2.1.1. 목적 및 개요

o Box Lab은 SmartX-mini Playground에서 오픈 소스 가상스위치 Open vSwitch[1]를 활용하여 서로 다른 종류의 호스트를 연결하는 과정을 담고 있다. 여기서 서로 다른 호스트란 물리적 머신, KVM [2] 기반의 가상머신(VM), Docker [3] 기반의 컨테이너를 의미한다. 오픈 소스 가상 스위치 OvS를 활용하여 이들 사이를 연결하고, 네트워크가 가능한 상태임을 확인한다.

o 본 기술문서에서 구성할 Box Lab의 환경은 [그림 1]과 같다. Intel 사의 NUC 미니 PC를 물리적 머신으로 활용하며, 물리적 머신에 오픈 소스 가상스위치 Open vSwitch를 활용하여 가상 브릿지 br0를 생성하고 VM과 컨테이너를 br0에 연결한다. VM 생성을 위해서 하이퍼바이저 KVM을 사용하고, Docker 기반의 컨테이너를 생성하여 이들 간의 통신이 가능하도록 연결한다. 연결이 완료된 후에는 각 호스트의 ip 주소로 ping 명령을 통해 연결을 확인한다.

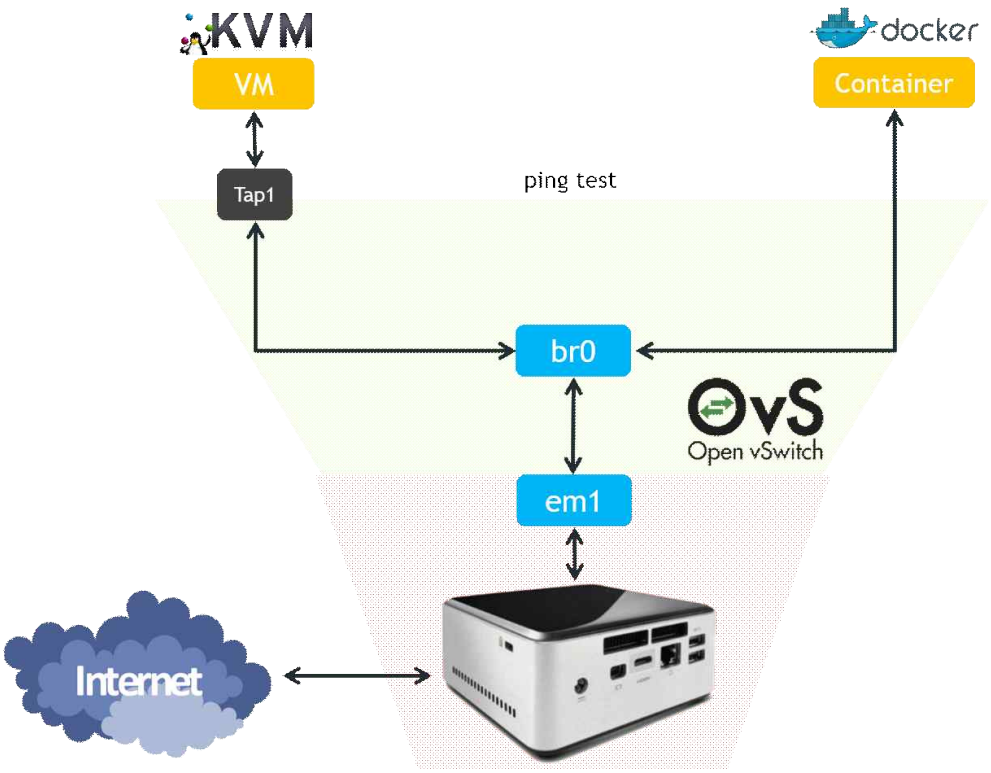


그림 3. Box Lab 구성 환경

2.1.2. Open vSwitch

o Open vSwitch는 리눅스 기반의 가상 소프트웨어 스위치로, Open Source Apache 2 License를 따른다. OpenFlow 프로토콜을 지원하며, SDN Switch로 사용할 수 있는 특징이 있다. OVSDB 프로토콜을 통해 설정값 등이 관리된다. 이 밖에도 터널링을 위한 GRE, VXLAN, IPsec 등을 지원한다. 모니터링을 위한 기능으로는 NetFlow, sFlow, IPFIX, SPAN, RSPAN 등을 지원하며, 현재 최신 LTS 버전으로는 2.5.1 버전이 릴리즈되었다. [1]

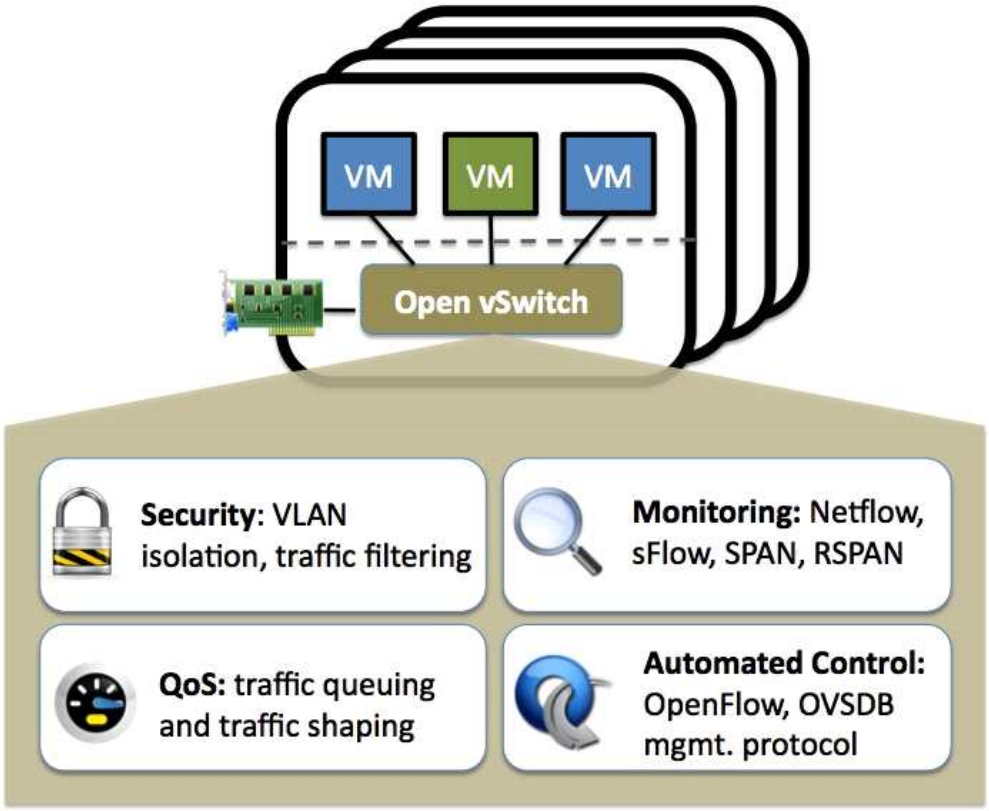


그림 4. Open vSwitch 개요

2.1.3. Kernel-based Virtual Machine(KVM)

o KVM이란 Kernel-based Virtual Machine의 약자로 호스트 컴퓨터에서 다수의 운영체제를 동시에 실행하기 위한 하이퍼바이저의 한 종류로써, [그림 3]과 같은 구조를 갖는다. KVM은 가상화를 제공하는 하이퍼바이저를 메모리 관리자나 파일 시스템 등과 같은 커널의 '서브 모듈'로 취급한다. KVM에서 가상화를 제공하기 위해서는 사용하는 H/W의 CPU에서 HVM(Hardware Virtual Machine) 기능을 제공해야한다는 전제가 있다. 최근 가상화 기능이 확산되면서 대부분의 CPU에서

해당 기능을 제공하고는 있으나, Box Lab 진행을 위해서 해당 사항을 확인해 볼 필요가 있다. [2]

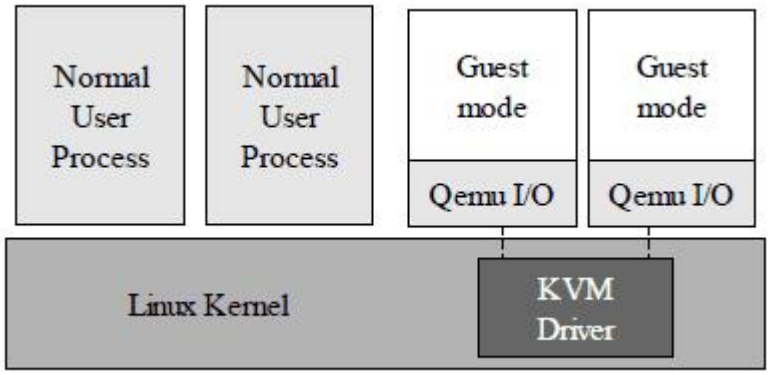


그림 5 KVM Hypervisor 아키텍처

2.1.4. Docker Container

o 도커(Docker)는 2013년에 등장한 새로운 컨테이너 기반 가상화 도구로써, 계층화된 파일시스템을 사용해 가상화된 컨테이너의 변경사항을 모두 추적하고 관리한다. 이를 통해 컨테이너의 특정 상태를 항상 보존해두고, 필요할 때 언제 어디서나 이를 실행할 수 있도록 도와주는 애플리케이션이다. [3]

o 가상 머신은 격리된 환경을 구축해준다는 점에서 매력적이지만, 실제 배포용으로 활용하기에는 성능 면에서 매우 불리한 도구일 수밖에 없다. 운영체제에서 또 다른 운영체제를 통째로 돌린다는 것은 자체로도 상당한 오버헤드를 발생시키기 때문이다. 도커 컨테이너 역시 이런 단점을 극복하기 위해서 나온 가상화 애플리케이션이다. 기존 가상 머신과는 달리 운영체제를 통째로 가상화하는 것이 아니라 커널 영역을 공유하면서 격리된 환경을 만들어주기 때문에 오버헤드가 크기 않고 유연하다.

2.2. 물리적 머신 (Physical Machine)

2.2.1. 물리적 머신 (Physical Machine) 환경 설정

o Box Lab 환경 구성을 위해 사용하는 물리적 머신은 Intel 사의 NUC miniPC다. 상세 스펙은 [표 1]과 같으며, 해당 머신에 Ubuntu 14.04.4 LTS Desktop 64bit 버전 (ubuntu-14.04.4-desktop-amd64)을 활용한다.

o Ubuntu 운영체제 설치가 완료되면, 가장 먼저 네트워크 인터페이스를 설정한다. Default로 설정되어 있는 네트워크 연결을 끊고, /etc/network/interfaces 파일을 열어서 [그림 4]와 같이 실험에 사용할 ip address, subnet mask, gateway, DNS

nameservers를 기재한다. 기재 후에는 터미널에서 ifdown, ifup, ifcoinfig 명령어를 통해 설정한 네트워크 인터페이스를 사용 가능하도록 만든다.

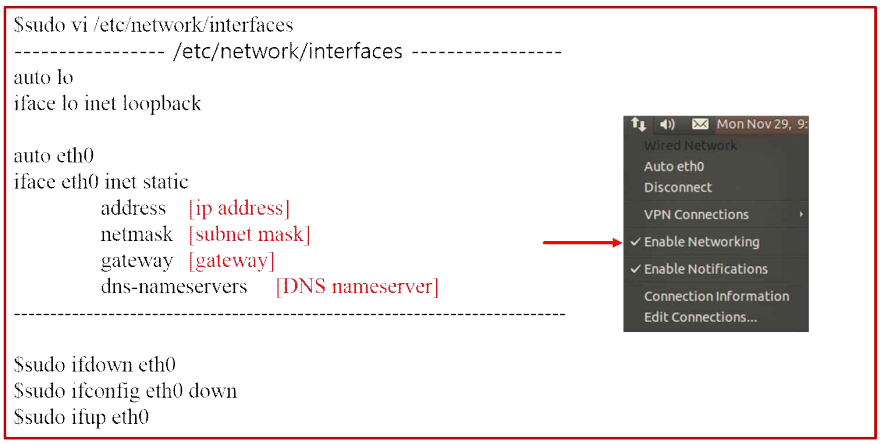


그림 6. 네트워크 인터페이스 설정

2.2.2. Open vSwitch 패키지 설치

o 네트워크 설정이 완료되었다면, 아래의 명령어를 통해 오픈 소스 가상스위치 Open vSwitch 패키지 설치를 위해 필요한 사전 작업을 진행한다. [그림 5]과 같이 패키지 목록을 업데이트하고 Open vSwitch 패키지를 설치한다.

#apt-get update

#apt-get install openvswitch-switch

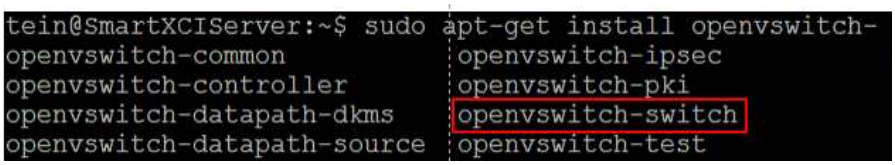


그림 7. 패키지 업데이트 및 Open vSwitch 설치

o Open vSwitch 패키지 설치가 완료되면, 각 호스트를 연결할 가상 브릿지 br0를 생성하고, br0에 ip를 할당하기 위해서 앞서 설정했던 네트워크 인터페이스를 [그림 6]처럼 변경한다. 이전에 eth0로 사용했던 네트워크 인터페이스를 br0로 변경한다.

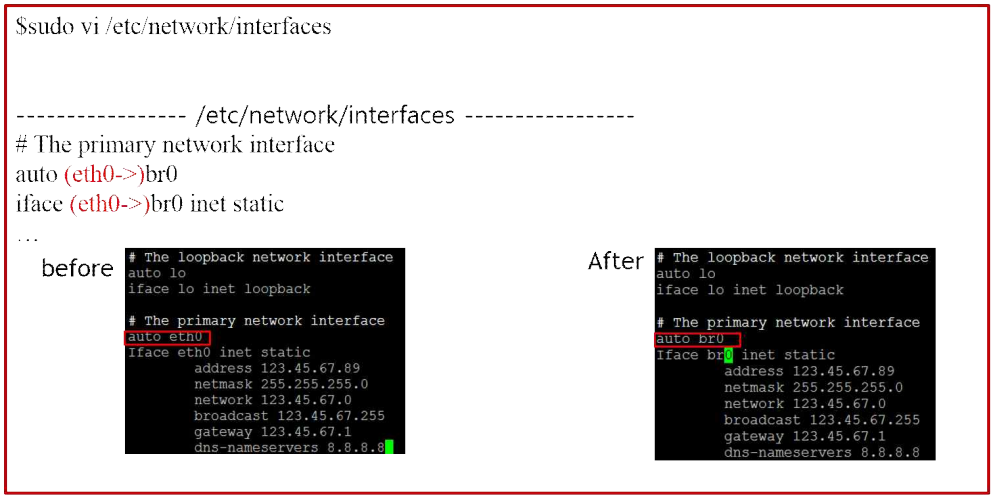


그림 8. 가상 브릿지 br0 생성을 위한 네트워크 인터페이스 설정

o 이어서 Open vSwitch를 활용하여 가상 브릿지 br0를 생성한다. 생성 후 정상적으로 생성이 완료되었는지 ovs-vsctl show 명령어를 통해 확인한다. 정상적으로 브릿지 생성이 완료되었다면 [그림 7]과 같은 화면을 확인할 수 있다.

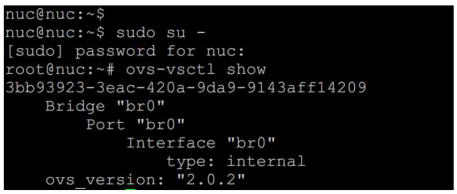


그림 9. 가상 브릿지 br0 생성 후 확인

o 이전에 설정했던 네트워크 인터페이스 eth0를 가상 스위치 Open vSwitch를 통해 생성한 가상 브릿지 br0에 연결해준다. ‘sudo ovs-vsctl add-port br0 eth0’ 명령어를 통해 가상 브릿지에 연결해주며, ‘sudo ifconfig eth0 0’, ‘sudo ifconfig br0 0’, ‘sudo ifup br0’, ‘sudo ifconfig br0 up’, ‘sudo ifconfig eth0 up’ 명령어를 통해 가상 브릿지에 ip 주소를 할당해준다. 이 과정에서 eth0 인터페이스가 global 영역에서 bridge 영역으로 이동하기 때문에 ssh와 같은 연결이 끊기므로, 직접 피지컬 머신에 모니터를 연결하여 수행할 것을 권장한다. 이 과정이 완료된 후 ifconfig 명령어를 통해 네트워크 인터페이스를 확인하면 [그림 8]과 같이 br0에 ip 주소가 설정된 것을 확인할 수 있다.

$sudo ovs-vsctl add-port br0 eth0

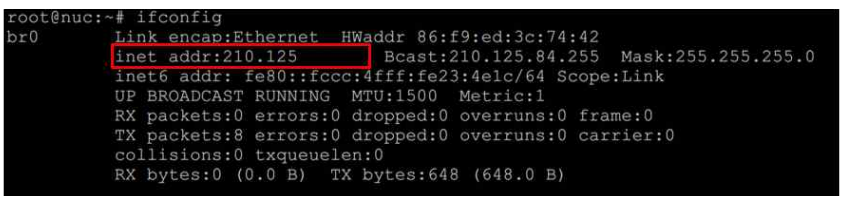


그림 10. 가상 브릿지 br0에 ip 주소가 할당된 화면

2.3. 가상 머신 (Virtual Machine)

2.3.1. KVM 패키지 설치, tap 설정 및 가상 머신 이미지 생성

o 물리적 머신(Intel Nuc miniPC)에서 VM 생성을 위한 하이퍼바이저를 설치한다. ‘sudo apt-get install qemu-kvm libvirt-bin’ 명령어를 통해 KVM 하이퍼바이저 설치에 필요한 패키지를 설치하고, VM에 사용할 Ubuntu 이미지 (ubuntu-14.04.3-server-amd64)를 다운로드 받는다.

o 아래의 명령어를 활용하여 가상 브릿지 br0에 가상 머신과의 네트워크 연결을 위한 tap을 설정해준다. [그림 9]의 모식도를 확인하면 현재까지 진행된 네트워크 연결 상황을 확인할 수 있다.

#ip tuntap add mode tap ‘tap_name’
#ifconfig ‘tap_name’ up
#ovs-vsctl add-port br0 ‘tap_name’

o 이어서 Virtual Machine 생성을 위해 아래의 명령어를 이용한다. 해당 명령어를 통해 10G 스토리지를 갖는 Ubuntu 14.04.3 server 64bit 버전의 Ubuntu 호스트 설치를 위한 이미지가 생성된다.

#qemu-img create ‘img_name’.img –f qcow2 10G
#kvm –m 512 –name vm1 –smp cpus=1,maxcpus=1 –device virtio-net-pci, netdev=net0, max=’EE:EE:EE:EE:EE:EE’ -netdev tap,id=net0,ifname=’tapname’,script=no, -boot d ‘img_name’.img –cdrom ubuntu-14.04.3-server-amd64.iso –vnc :2 -daemonize

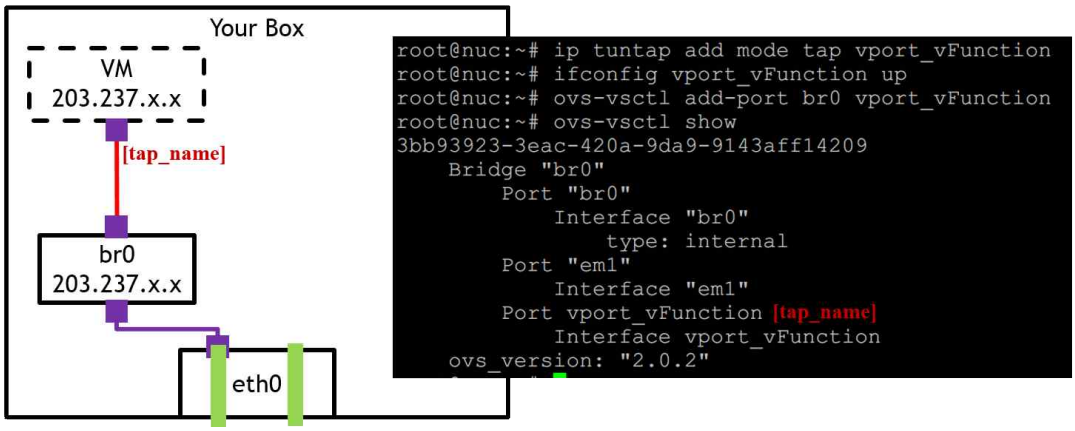


그림 11. 가상 브릿지 br0에 tap 연결

o 설정을 위해 기존에 사용중인 물리적 머신이 아닌 다른 호스트에서 VNC Viewer를 통해 가상 머신 내에 Ubuntu OS 설치를 진행한다. VNC Viewer에서 [그림 10]과 같이 설정함으로써 가상 머신에 접근할 수 있다.

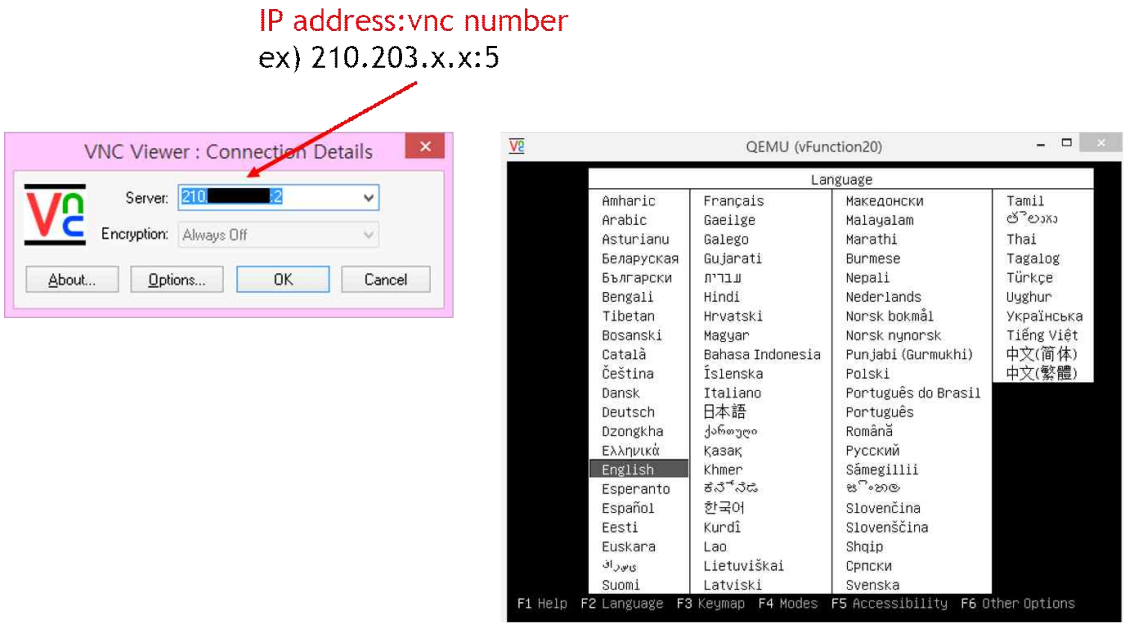


그림 12. VNC Viewer를 통한 가상머신 접근

2.3.2. Ubuntu OS 설치 과정

o VNC Viewer를 통해 가상 머신 OS 설치화면에 접근했다면 [그림 11]과 같은 보라색 화면을 확인할 수 있다. 이 화면에서 개인의 세팅에 따르는 값(ip address, gateway address, dns-nameservers address 등)을 입력하여 운영체제 설치를 진행한다.

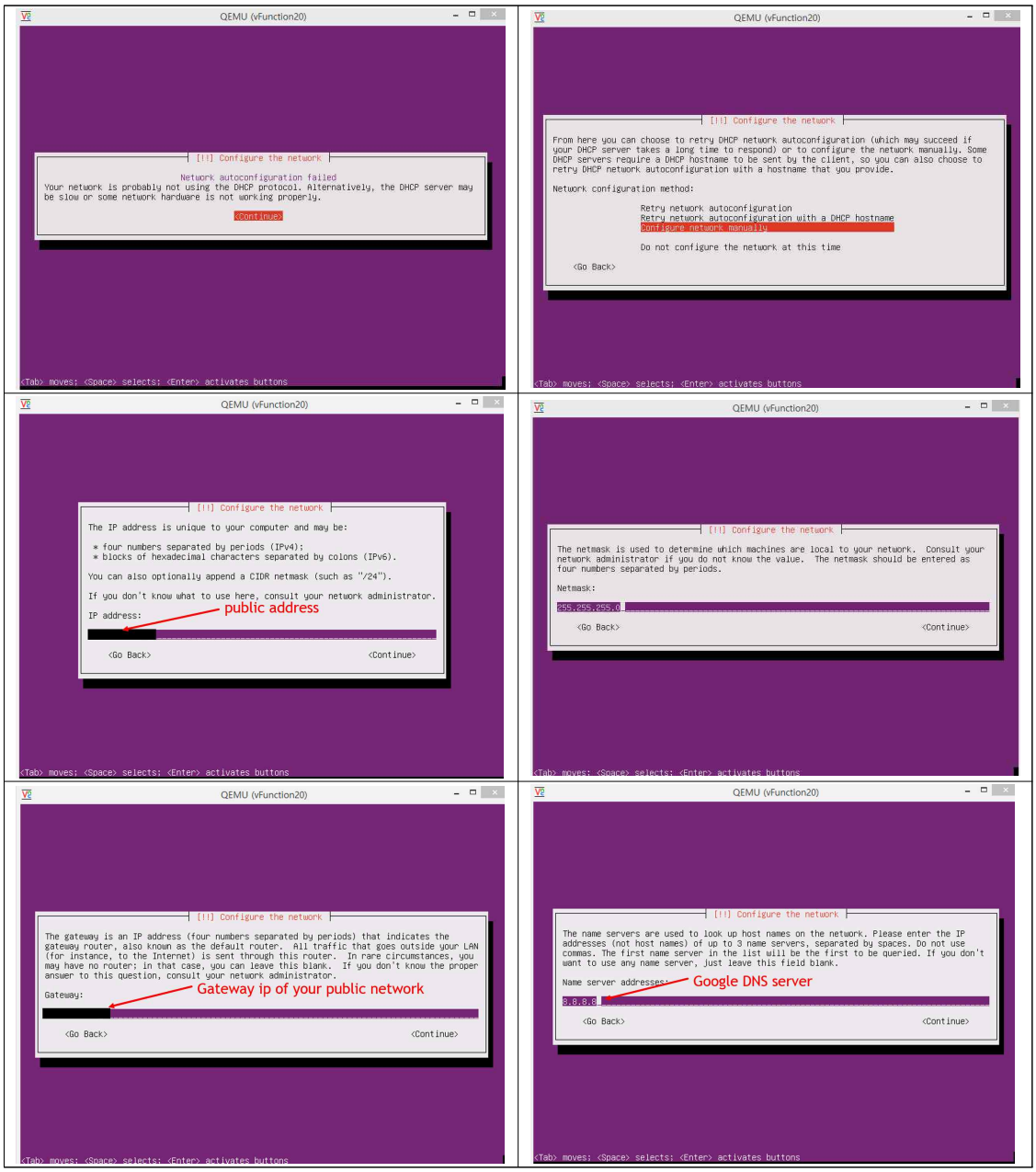


그림 11. 가상 머신 내 Ubuntu OS 설치 화면

o 설치가 완료된 후 [그림 12]와 같은 화면으로 가상 머신이 부팅되면 'Boot from first hard disk' 버튼을 통해 설치된 Ubuntu OS의 호스트로 접속한다.

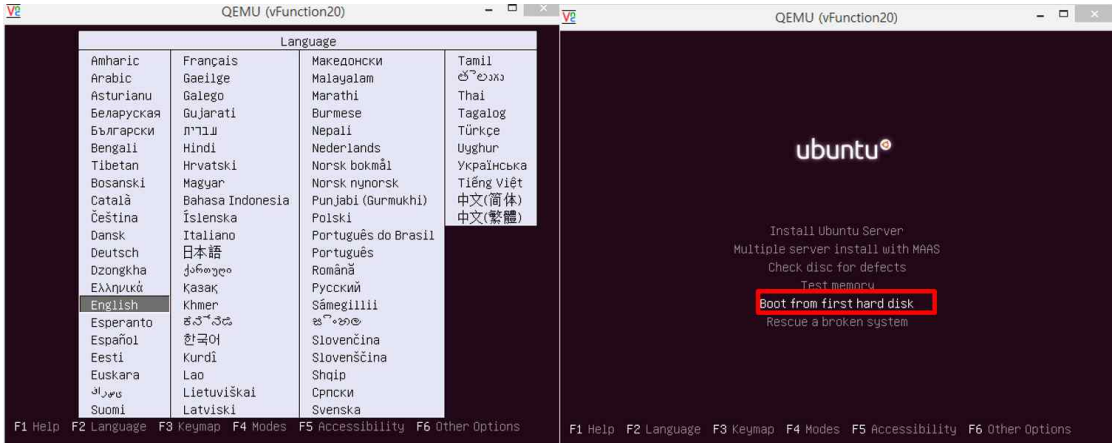


그림 12. 가상 머신 내 Ubuntu OS 설치 후 접속 화면

2.4. Docker 컨테이너 (Docker Container)

2.4.1. Docker 설치

o 아래의 명령어를 통해 웹으로부터 docker의 최신버전을 설치할 수 있다. 설치 후 컨테이너 제작을 위해 Docker 서비스를 실행하고, 서비스에 접근할 유저를 생성한다. 정상적으로 설치가 완료되었는지 확인을 위해 기본적으로 설정되어 있는 'hello-world' 컨테이너를 실행해본다. 정상적으로 설치가 진행되고 있다면, [그림 13]과 같은 화면을 확인할 수 있다.

#wget -qO- https://get..docker.com/ | sh
#service docker start
#adduser 'user_id' docker
#docker run hello-world

그림 13. 'hello-world' 컨테이너 실행화면

2.4.2. 컨테이너 (Container) 생성

o 본 기술문서에 기재된 Docker 설치 과정에 따라 정상적으로 설치 과정이 완료되었다면, 이어서 아래의 명령어를 활용하여 Docker 기반의 컨테이너를 생성한다. 명령어에 따라 컨테이너가 생성되면서 [그림 14]와 같이 root 계정으로 쉘 커맨드 입력이 진행되는 것을 확인할 수 있다. 이는 생성된 컨테이너 내부 쉘 커맨드라인으로 정상적으로 컨테이너가 생성되었음을 의미한다.

#docker run -it —net=none —name 'container_name' ubuntu /bin/bash

```
nuc@nuc:~$ docker run -it --net=none --name c1 ubuntu /bin/bash
root@8346684676d8:/#
```

그림 14. 컨테이너 생성 후 내부 쉘 커맨드라인 화면

2.4.3. Open vSwitch와 Docker 컨테이너 연결

o Box lab 실험에 사용할 Docker 컨테이너 생성이 완료되었다면, 이제 물리적 머신(Intel NUC Mini PC)에 설치된 오픈 소스 가상 스위치 Open vSwitch를 사용하여 네트워킹이 가능하도록 컨테이너와 가상 브릿지 br0를 연결시켜준다. 이 과정에서 'ovs-docker'라는 유틸리티가 필요하다. 해당 유틸리티를 다운로드 받은 뒤 아래의 명령어를 통해 Open vSwitch 브릿지 br0와 Docker Container를 연결시킨다.

$cd /usr/bin
#wget https://raw.githubusercontent.com/openvswitch/ovs/master/utilities/ovs-docker
#sudo chmod a+rwx ovs-docker
#docker attach 'container_name'

o Docker를 활용하여 컨테이너를 생성하면, 컨테이너 내부에는 네트워킹에 필요한 기본적인 패키지들이 포함되어 있지 않다. 따라서 아래의 명령어를 통해 네트워킹 기능 검증에 필요한 ping 기능이 가능하도록 패키지를 업데이트 및 설치한다.

#apt-get update
#apt-get install net-tools
#apt-get install iputils-ping

o 패키지 업데이트 및 설치가 완료되면 ping 테스트를 통해서 외부와의 네트워킹이 가능한지 확인해본다. 정상적으로 설정이 마무리되었다면 외부로의 네트워킹이 가능하다.

#ping google.com

2.5. Box Internal Networking 검증

2.5.1. 시나리오에 따른 네트워킹 확인

o 컨테이너의 세팅이 완료되면서, 1장의 [그림 1]과 같이 하나의 물리적 머신 위에 하이퍼바이저를 이용한 가상머신, 그리고 Docker 컨테이너가 모두 연결된 환경이 구성되었다. 이제 최종적으로 아래의 세 가지 시나리오를 검증함으로써 정상적으로 네트워킹이 가능한 지 확인한다.

i. 가상 머신에서 박스 외부로의 네트워크로 ping 테스트 확인
ii. 컨테이너에서 박스 외부로의 네트워크로 ping 테스트 확인
iii. 가상 머신에서 컨테이너로 박스 내부 네트워킹 ping 테스트 확인

o 위 세 가지 경우에 대해서 이상 없이 ping 테스트 검증이 완료되었다면, SmartX-Labs의 첫 시작인 Box Lab이 성공적으로 진행된 것이다. 정상적으로 테스트된 화면은 [그림 15]와 유사하게 출력될 것이다.

o Box Lab의 경우 크게 세 가지 컴포넌트로 구성된다. 물리적 머신, 가상 머신 그리고 컨테이너. 만약 위 테스트에서 일부 동작하지 않는 시나리오가 있다면, 해당하는 컴포넌트의 설명 자료를 토대로 설정을 재수정하여 진행하는 것을 권장한다.

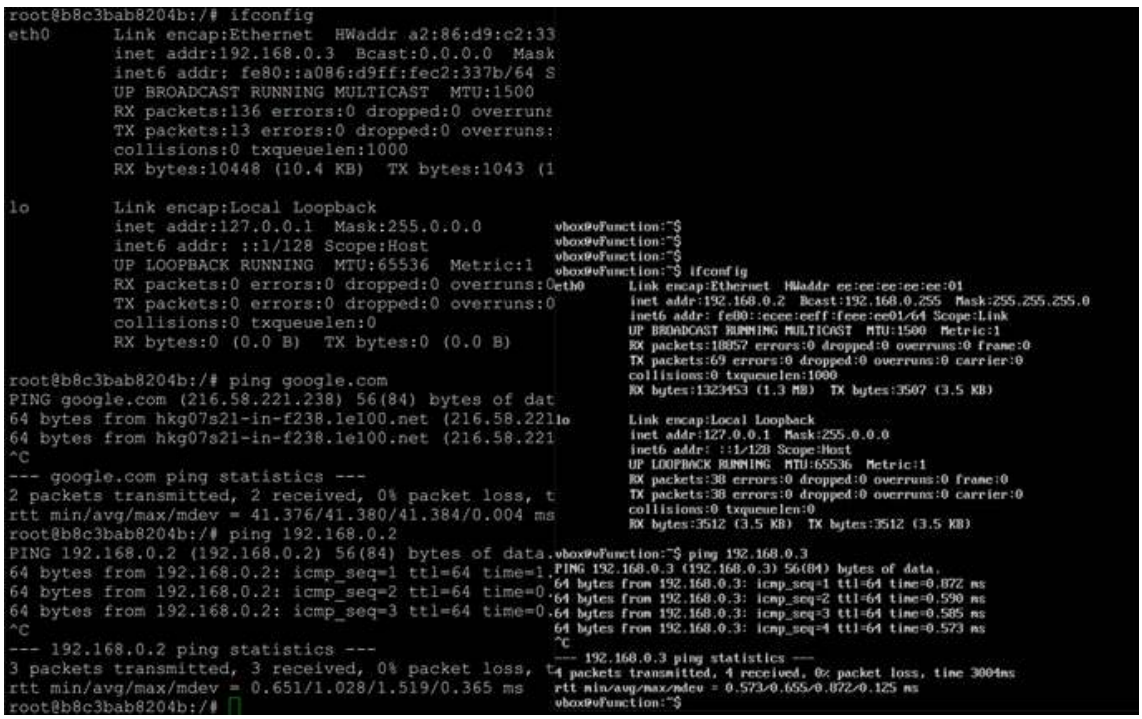


그림 15. 가상 머신의 네트워크 인터페이스, 컨테이너의 네트워크 인터페이스와 이들 사이의 네트워킹 기능(ping test) 검증 화면

2.6. 결론

o 본 기술문서는 리눅스 기반의 OS가 설치된 피지컬 머신에서 KVM 하이퍼바이저를 통한 가상 머신을 생성하여 내부적으로 격리된 공간의 별개의 호스트를 생성해본다. 그리고 Docker 애플리케이션을 통해 커널 영역은 공유하지만 또 다른 형태의 격리된 호스트 컨테이너를 생성해본다. 그리고 오픈 소스 가상 스위치 Open vSwitch를 활용하여 가상 브릿지 br0를 생성해보고, br0에 가상 머신과 컨테이너를 연결함으로써 그들 사이의 네트워킹이 가능하도록 한다.

o 본 기술문서를 통해 가상화에 익숙하지 않은 학생 혹은 비-전공자들로 하여금 두 가지 다른 형태의 가상화 기술을 사용하여 격리된 호스트를 생성해보고, 가상 스위치를 통해 이들을 네트워크로 엮어봄으로써 가상화, 그리고 오픈소스 가상 스위치에 대한 기본적인 기능 및 사용환경을 경험해볼 수 있도록 한다.

3. InterConnect Lab

3.1. InterConnect Lab Background

3.1.1. 목적 및 개요

o InterConnect Lab은 이전 Box Lab에서 구축한 Box에 다른 기기를 연결하고, 연결된 기기에 대한 리소스 정보들을 확인한다. 연결된 기기에 대한 리소스 정보 확인을 위해서 SNMP, Flume, Kafka등을 사용하므로 이에 대한 이해가 필요하며, 이를 활용해 연결된 기기에 대한 리소스 정보를 수집하고 상태 정보 데이터를 확인한다.

o 본 기술문서에서 InterConnect Part의 구성환경은 그림 1과 같다. 이전 Lab에서 구축한 Box에 라즈베리파이라고 하는 싱글보드 컴퓨터와 연결을 한다. 이후 Net-SNMP와 Flume을 통해 라즈베리파이에 대한 리소스 정보를 전송하고 구축한 Box에서는 Kafka를 활용해 전송된 데이터를 수집한다. 마지막으로 수집된 데이터를 최종적으로 확인해보는 과정을 다뤄본다.

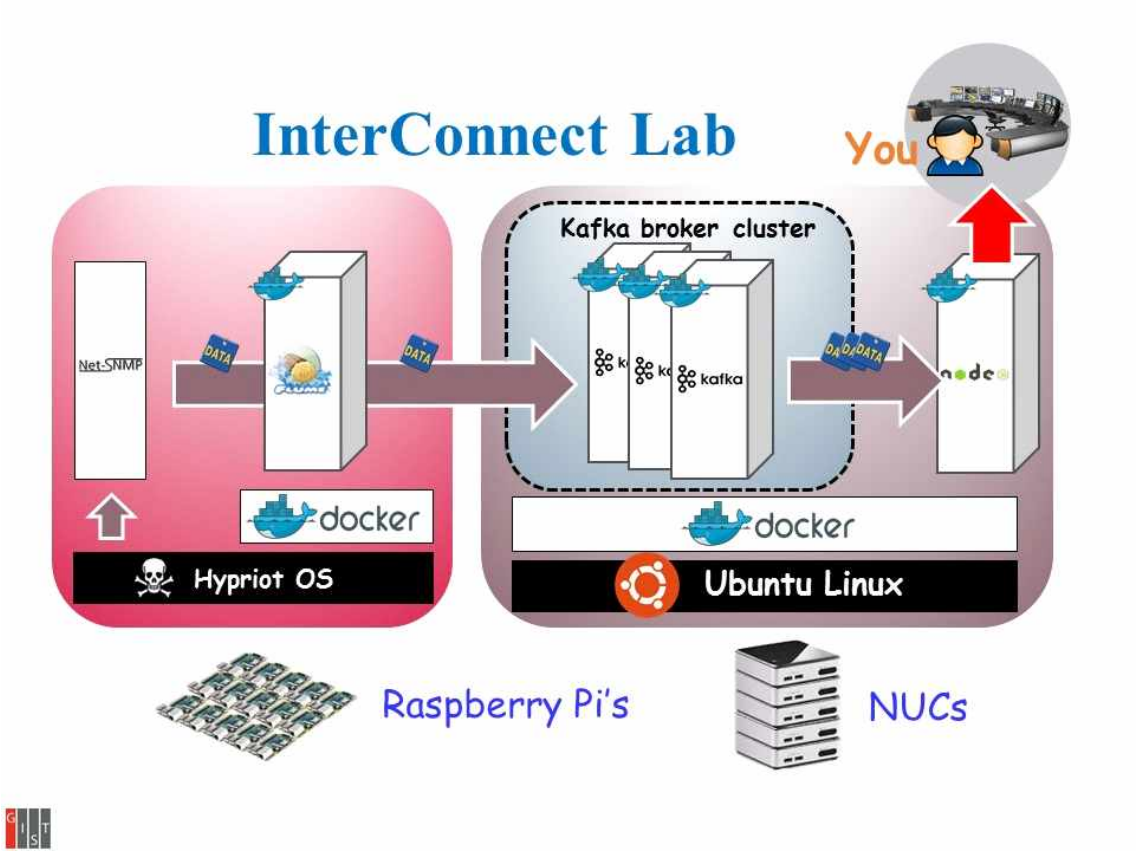


그림 23 InterConnect Lab 구성환경

3.1.2. SNMP / Net-SNMP(Simple Network Management Protocol)

o SNMP(Simple Network Management Protocol)는 네트워크 운영을 위한 보편적

인 프로토콜이다. 네트워크로 연결된 장치에 대한 리소스 정보 및 상태 정보를 확인하는데 사용되며, 단순히 PC 뿐만 아니라 서버, 허브, 스위치, 그리고 라우터 등 IP(Internet Protocol) 네트워크로 연결된 장치들에 대해 사용이 가능하다.

o SNMP를 사용하기 위해서는 Net-SNMP라는 소프트웨어를 설치해야 한다. Net-SNMP는 연결된 기기들을 SNMP agent와 SNMP manager로 두 부류로 나눈다. SNMP 소프트웨어를 사용해 SNMP manager는 SNMP agent로 지정된 장치의 상태 정보를 받아올 수 있고, 이를 통해 모니터링이 가능하다. 그림 2는 Net-SNMP에서 SNMP를 사용해 상태 정보를 받아오는 과정을 나타낸 그림이다. SNMP Manager에서 주황색 칸에 들어간 함수를 통해 SNMP agent에 요청을 하고, SNMP agent는 요청에 대한 답과 상태 정보를 응답으로 전송한다.

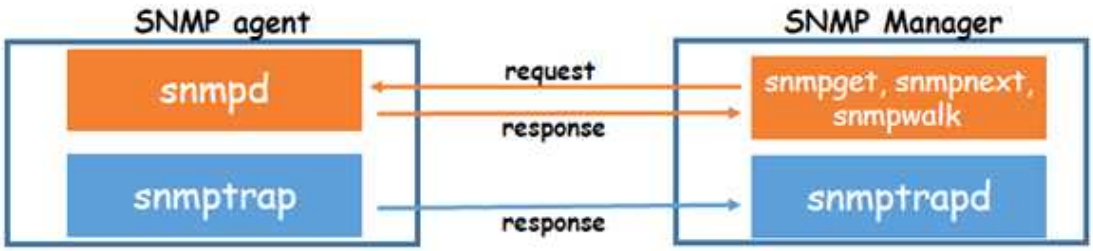


그림 2 Net-SNMP

3.1.3. Flume

o Flume은 수많은 서버에 분산되어 있는 많은 양의 로그 데이터를 한 곳으로 모이게 해 주는 로그 수집기이다. 수많은 서버의 로그 데이터가 다양하게 커스터마이즈된 데이터일 경우이거나 장애가 발생하더라도 로그를 유실 없이 전송해 줄 필요가 있고, 분산된 서버에서의 데이터 수집을 위해 분산 수집 구조가 가능해야 한다. 이를 위해 Apache에서는 2011년에 Cloudera CDH3에서 Apache Flume을 공개했다.

o Flume을 이해하기 위해서는 크게 Event와 Agent를 알아야 한다. Flume에서 Event는 Flume을 통해 전달되어지는 데이터의 기본 Payload를 지칭한다. Agent는 크게 Source, Channel, Sink로 나눌 수 있고, 이는 그림 3으로 확인할 수 있다. Source는 Event를 수집해서 Channel로 전달하는 역할을 하는데, 추가로 Source내의 기능인 Interpretor, Channel Selector를 통해 수집된 데이터를 변경하거나 Channel을 지정할 수 있다. Channel은 이름 그대로 Event를 Source와 Sink로 전달하는 통로이며, 마지막으로 Sink는 Channel로부터 받은 Event를 제거하거나 외부의 저장소나 또 다른 Source로 전달한다.

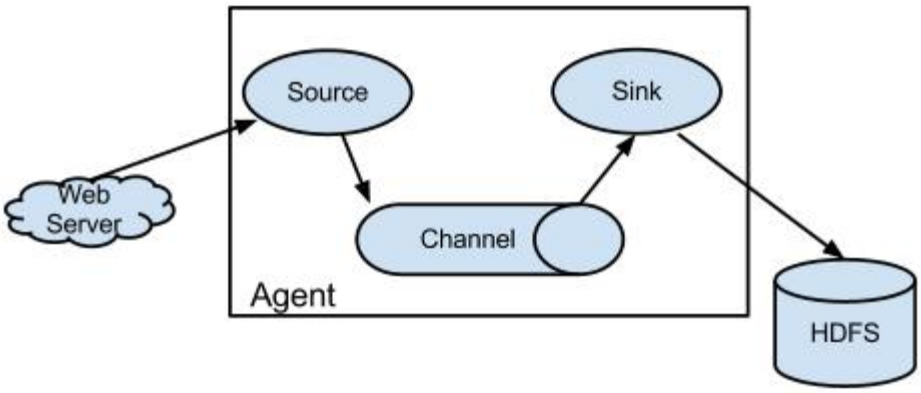


그림 3 Apache Flume Agent

3.1.4. Kafka

o Kafka는 아파치 소프트웨어 재단이 Scala로 개발한 오픈 소스 메시지 브로커 프로젝트이다. 비즈니스, 소셜 등 다양한 애플리케이션에 의해 실시간 정보가 끊임없이 생성되는데 생성된 실시간 정보를 많은 수신자에게 전달해야 하는데 정보 생산 애플리케이션과 수신 애플리케이션이 분리가 되어 있어서 서로 접근이 불가능했고, 이런 문제를 해결하기 위한 프로젝트이다.

o 높은 처리량은 물론이고, 정체 현상과 과부하를 막기 위해 파티셔닝을 지원해 다수의 카프카 서버에서 분산 처리가 가능하게 했다. 또한 정보 생산과 수신 애플리케이션의 분리로 인한 연동의 문제를 해결하기 위해 다양한 플랫폼과도 연동이 되게 했으며, 실시간 처리가 가능하다.

o Kafka는 메시지를 카테고리로 나눈 후 메시지를 처리하는데, 해당 카테고리를 topic이라고 한다. 메시지를 분산 처리하는데 Kafka는 크게 Producer, Broker, Zookeeper, Consumer로 나눌 수 있는데, 이들은 발행-구독(publish-subscribe) 모델을 기반으로 동작하고 이는 그림 4에 나타나 있다. Producer에서는 특정 topic의 메시지를 생성한 뒤 해당 메시지를 broker로 전송한다. broker는 Producer로부터 전송 받은 메시지를 topic에 기준해 관리하고, 데이터를 계속 쌓아둔다. 이후 해당 topic을 구독하는 Consumer는 broker에 쌓인 메시지를 처리한다. Kafka는 확장성과 고가용성을 위해 broker가 클러스터 형태로 동작하게끔 설계를 했고 클러스터화된 broker를 관리하기 위해 Zookeeper가 사용된다.

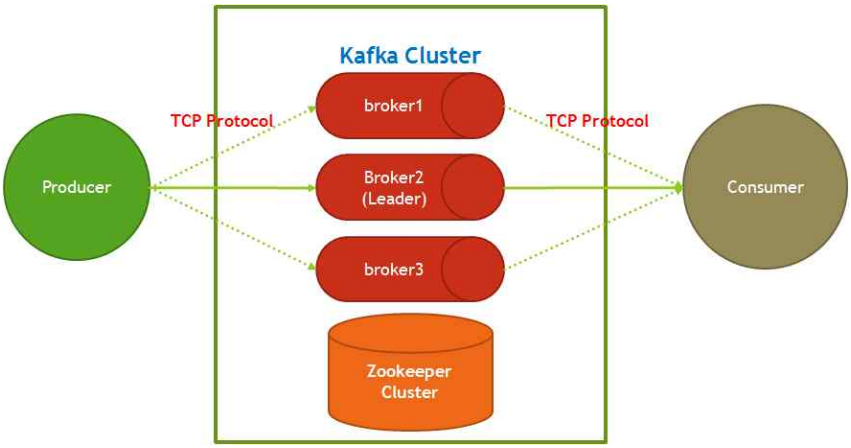


그림 4 Kafka Architecture

3.2. Settings & Configuration

3.2.1. NUC 설정

o 우선 Github에 저장된 파일을 다운받아 오면 다음과 같은 폴더가 하나 생성된다. 그림 5에 있는 폴더명과 동일하다면 성공적으로 다운로드 된 것이다.

$ docker build --tag ubuntu-kafka .

```
root@ubuntu:/home/chorwon# git clone https://github.com/SmartXBox/SmartX-mini.git
Cloning into 'SmartX-mini'...
remote: Counting objects: 5935, done.
remote: Total 5935 (delta 0), reused 0 (delta 0), pack-reused 5935
Receiving objects: 100% (5935/5935), 19.05 MiB | 1.53 MiB/s, done.
Resolving deltas: 100% (1412/1412), done.
Checking connectivity... done.
root@ubuntu:/home/chorwon# ls
SmartX-mini
```

그림 5 Github에서 다운로드 한 결과

o 다운 받은 후 SmartX-mini라는 경로로 이동을 하면, 다음과 같이 4개의 폴더가 있고, 폴더명은 그림 6와 같다. 여기서는 ubuntu-kafka로 접근한다.

그림 6 SmartX-mini 디렉토리

o ubuntu-kafka라는 폴더로 이동 후에는 IP address table를 정의해야 한다. 1개의 Zookeeper와 3개의 broker를 활용하기 때문에 각각에 대한 IP 주소를 지정해야 한다. 그림 6에서는 IP address table에 대한 예시를 보여주고 있는데, 해당 표에서 IP address 부분을 제외한 나머지 부분은 고정해 주는 것이 앞으로의 실험을 위해서 편하다. 또한 IP address는 5개가 필요하다.

Host Name	IP address	Broker id	Listening port
zookeeper	192.168.0.x	-	2181
broker0	192.168.0.x	0	9092
broker1	192.168.0.x	1	9092
broker2	192.168.0.x	2	9092

그림 7 정의된 IP address table

o ubuntu-kafka 폴더 안에는 Dockerfile이 있는데 이는 실험을 위해 kafka를 사용하기 위해 설정한 Docker image를 생성하기 위한 스크립트이다. 이를 활용해 Docker image를 빌드하고 실행시킨다.

```
$ docker build --tag ubuntu-kafka .
```

o 빌드된 Docker image를 활용해 Docker Container를 실행할 수 있다. 실행하는 방법을 다음과 같다.

```
$ docker run -it --net=none -h [host name] --name [container name] ubuntu-kafka
```

여기서 쓰인 docker run의 매개변수의 의미는 다음과 같다. -it는 컨테이너와 상호적으로 주고받으면서, 터미널과 비슷한 환경을 조성해주는 의미이다. -h와 --name은 호스트 이름과 Container 이름을 지정하는 부분으로 호스트 이름은 그림 6을 활용해 정의한 host name을 사용하면 된다.

o 이전에 --net=none이란 명령어를 사용해서 컨테이너를 실행했기 때문에, 당연히 네트워크 연결이 된 상황이 아니다. 이전 Box Lab에서 br0라고 하는 브릿지를 만들어 놨기 때문에 컨테이너를 브릿지로 직접 연결을 시켜줘야 한다. 이에 대한 명령어는 다음과 같다.

```
$ sudo ovs-docker add-port br0 eth0 [container name] --ipaddress=[container ip address]/24 --gateway=[gateway address]
```

o 이후 다음 명령어를 활용해 컨테이너 내부로 들어가서 hosts라는 파일을 그림 7을 활용해 수정한다. 여기에 들어간 것은 IP 주소와 hostname으로 환경에 맞게 수정해서 저장한다.

$docker attach [container name]
$ sudo vi /etc/hosts

```
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

210.125.88.10 zookeeper1
210.125.88.20 broker0
210.125.88.21 broker1
210.125.88.22 broker2
```

그림 8 /etc/hosts 설정 예시

o Kafka의 클러스터링에는 Zookeeper와 Broker, 두 개가 활용되기 때문에 이 둘이 서로 통신을 할 수 있게 IP 주소를 설정하면서 포트 번호를 동일하게 해줘야 한다. 포트 번호는 디폴트 설정을 그대로 사용할 것이기 때문에 변경해 줄 필요는 없으나, broker의 설정이다. broker에서 Zookeeper의 주소를 자신의 환경에 맞게 바꿔 준다. 또한 broker 각각의 Id는 겹치면 안 되므로, 3개의 broker의 id는 모두 다르게 해야 한다. 이는 그림 9에 나와 있다.

$vi config/server.properties

```
############################ Server Basics #

# The id of the broker. This must be set to a
broker.id=0

############################ Socket Server S

# The port the socket server listens on
port=9092
```

```
############################ Zookeeper ###

# Zookeeper connection string (see zookeepe
# This is a comma separated host:port pairs
# server. e.g. "127.0.0.1:3000,127.0.0.1:30
# You can also append an optional chroot st
# root directory for all kafka znodes.
zookeeper.connect=localhost:2181
```

그림 9 broker 설정 변경

그림 9의 왼쪽 그림의 broker.id 값은 3개의 broker가 모두 달라야 하며, 오른쪽의 zookeeper.connect 값은 localhost 부분을 지운 후, 이전에 정의한 IP address table에서 지정된 zookeeper IP address로 바꿔준다.

o 위의 과정을 1개의 Zookeeper와 3개의 broker 모두 해줘야 하기 때문에 총 4번

수행해야 한다.

o 4번의 반복 작업 후에는 1개의 Zookeeper Container와 3개의 broker Container가 생성됨을 확인할 수 있다. 이후, Zookeeper와 Broker를 실행시키는데 반드시 Zookeeper를 먼저 시작해야 한다. 이는 그림 10과 11을 통해 확인 가능하다.

$bin/zookeeper-server-start.sh config/zookeeper.properties

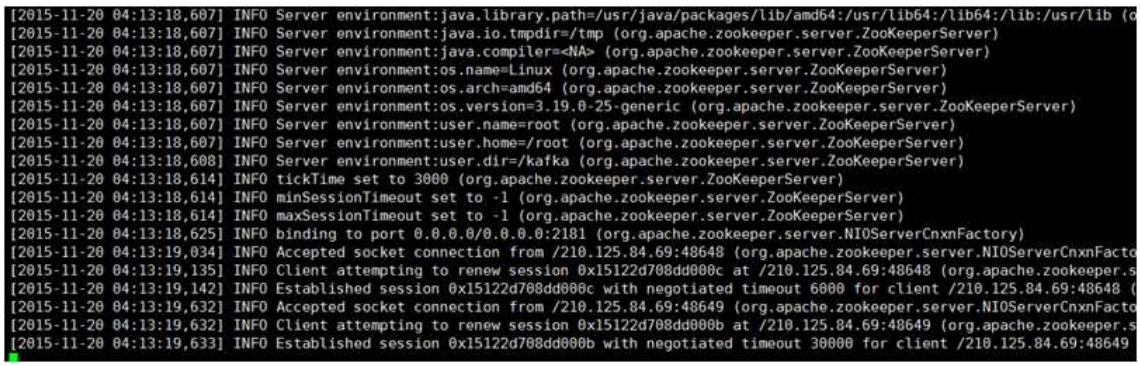


그림 10 Zookeeper 실행 결과

$bin/kafka-server-start.sh config/server.properties

그림 11 broker 실행 결과

o Zookeeper와 broker를 만든 후에는 메시지에 대한 topic 설정 및 Consumer를 만들어야 한다. 우선, Consumer는 다음의 명령어를 활용해 컨테이너로 생성한다.

$docker run -it --net=host --name [container name] ubuntu-kafka

이후, Consumer의 컨테이너의 /etc/hosts 파일 이전에 zookeeper와 broker에서 설정하는 방법과 동일하게 수정한다.

o Consumer까지 설정이 끝났다면, topic을 만든다. topic은 Consumer 컨테이너 내부에서 생성을 해야 하고, topic을 만드는 명령어는 다음과 같다.

$ bin/kafka-topics.sh --create --zookeeper [zookeeper host name]:2181
--replication-factor 1 --partitions 3 --topic <topic_name>

3.2.2. Raspberry Pi

o 지금까지 리소스 상태 정보를 받아서 보여주는 부분을 만들었다면, 이제는 리소스 상태 정보를 보내주는 부분을 만들어야 한다. 이를 위해서 싱글보드 컴퓨터인 라즈베리파이를 활용한다. 라즈베리파이는 32 bit의 운영체제를 사용하는데, Docker 컨테이너를 활용하기 위해서는 64 bit의 운영체제가 필요하다. 따라서 원래는 라즈베리파이에서 컨테이너를 활용할 수가 없지만, 다행히도 이 문제점을 해결하기 위한 라즈베리파이 운영체제인 Hypriot OS가 있고 해당 운영체제를 활용할 것이다. Hypriot을 라즈베리파이에 설치하기 위해서는 http://blog.hypriot.com/downloads/로 접속 한 후, 이미지 파일을 다운 받는다. 이후 SD Writer을 다운 받은 후, 다운 받은 Hypriot 이미지를 SD card에 설치한 후 설치된 SD card를 라즈베리파이에 꽂아주면 된다.

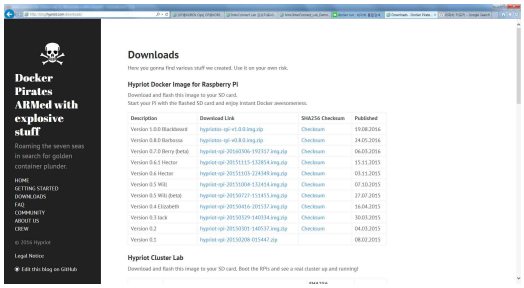


그림 12 Hypriot 공식 홈페이지

o Hypriot이 설치된 라즈베리파이에 SNMP를 활용하기 위한 Net-SNMP 및 kafka를 설치할 때와 마찬가지로 github에서 SmartX-mini를 다운 받은 후 raspbian-flume에 대한 Dockerfile을 빌드한다. Net-SNMP를 설치하는 과정은 다음과 같다.

```
$ sudo apt-get update
$ apt-get install -y snmp snmpd
$ apt-get install -y snmp-mibs-downloader
$ download-mibs
```

Net-SNMP를 설치한 후에는 snmpd.conf라는 설정 파일에서 #rocommunity public localhost에서 #을 빼준다. 결과는 그림 13처럼 될 것이다.

그림 13 snmpd.conf 설정

다음으로는 raspbian-flume에 저장된 Dockerfile을 빌드한다. Dockerfile은 이전의 Kafka Dockerfile과 마찬가지로 Flume을 사용하기 위해 설정된 Docker image를 빌드하기 위한 스크립트이다. 빌드 후에는 docker run 명령어를 활용해 실행해야 하지만, 실행 이전에 /etc/hosts 파일을 설정해 줄 필요가 있다. 해당 설정은 이전에 정의한 IP address table을 활용해 설정한 방법과 동일하다. 그림 14에 첨부된 예시를 활용해 자신의 환경에 맞게 설정한다.

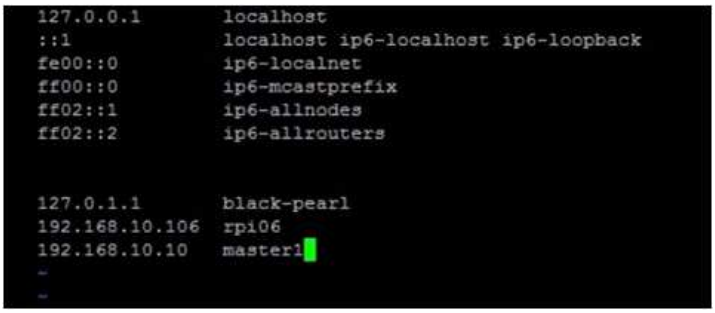


그림 14 라즈베리파이에서의 /etc/hosts 설정 예시

이후에는 docker run을 통해 Flume을 실행한다. 이에 대한 내용은 다음과 같다.

```
$ cd SmartX-mini/raspbian-flume
$ docker build --tag raspbian-flume .
$ docker run -it --net=host raspbian-flume
```

Zookeeper와 broker에 대한 설정을 환경에 맞게 바꾸듯이 Flume에 대한 설정을 바꿔 줄 필요가 있다. 바꿔야 할 설정 파일은 conf/flume-conf.properties로 여기서는 agent.sinks.sink1.topic의 값을 이전에 지정한 topic의 이름으로 변경해 줘야 하고, agent.sinks.sink1.brokerList의 값은 broker의 ip 주소와 포트 번호를 써줘야 한다. topic의 이름과 broker ip 주소 및 포트 번호는 자신의 환경에 맞게 수

정을 하고, 그에 대한 결과는 그림 15와 동일해야 한다.

```
# The sink
agent.sinks.sink1.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.sink1.topic = topic1
agent.sinks.sink1.brokerList = broker1:9092,broker2:9092,broker3:9092
agent.sinks.sink1.requiredAcks = 1
agent.sinks.sink1.batchSize = 1
```

그림 15 수정 후의 conf/flume-conf.properties

o Flume Agent까지 설정을 다 완료했다면, 이제 Net-SNMP와 Flume을 활용해 broker로 메시지를 보내줘야 kafka에서 메시지를 처리한 후, 라즈베리파이에 대한 상태 정보를 확인할 수 있다. 따라서 Flume을 실행시킨다.

```
$ bin/flume-ng agent --conf conf --conf-file conf/flume-conf.properties
--name agent -Dflume.root.logger=INFO,console
```

3.2.3. 결과

o 이제 설정한 라즈베리파이에서 전송되는 라즈베리파이에 대한 상태 정보 및 리소스 정보를 NUC에서 확인하면 된다. 이를 위해 NUC에서 만들어 놓은 Consumer 컨테이너로 들어가서 다음의 명령어를 활용한다. 이후, 그림 16과 같이 비슷한 형태로 Consumer 컨테이너에서 출력된다면 끝이 난다.

```
$bin/kafka-console-consumer.sh --zookeeper [zookeeper host name]:2181
--topic [topic name] --from-beginning
```

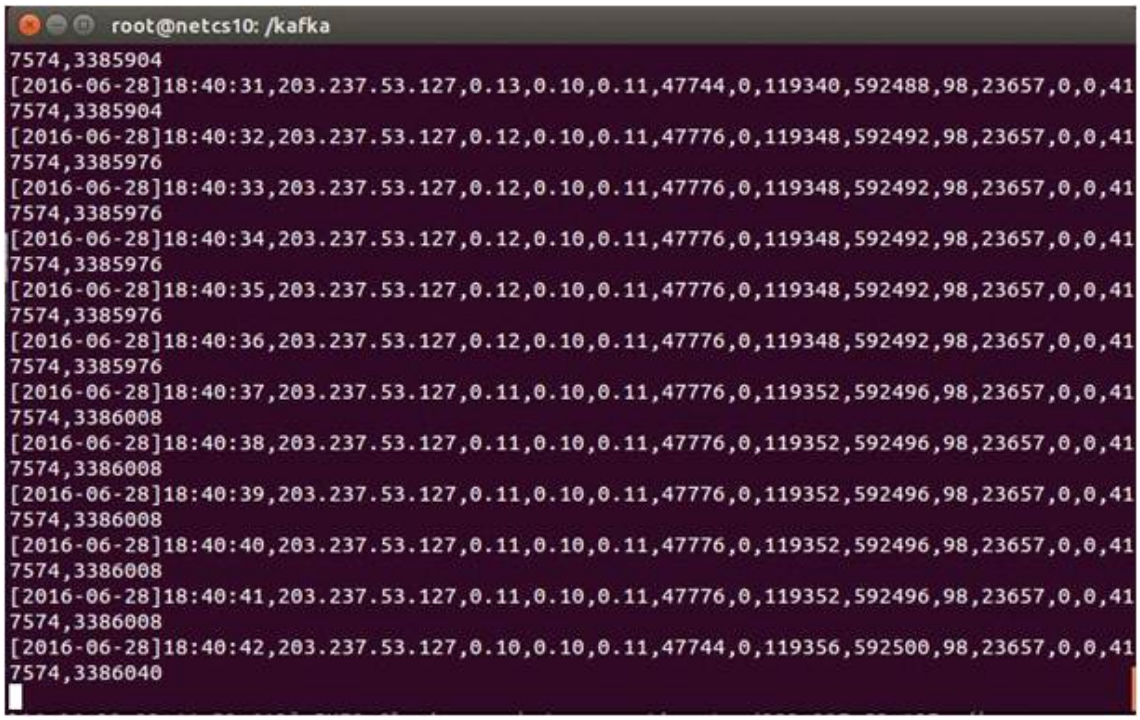


그림 16 최종 결과

o 결과를 확인하면, 맨 앞에는 해당 날짜와 시간이 나온다. 이후에는 라즈베리 파이에 대한 IP 주소가 나오면서, 리소스 상태에 대해 모니터링 중인 값이 차례로 나온다.

3.3. 결론

o 본 기술문서의 InterConnect Labs 단원은 IoT-Cloud 환경에 대해 생소한 사람들을 대상으로 작성되었으며 IoT 디바이스인 라즈베리파이에서의 리소스 및 상태 정보를 마이크로 Cloud 환경인 NUC에서 확인하기 위해 SNMP와 Flume과 Kafka를 활용하는 식으로 구성된다.

4. Tower Lab

4.1. Tower Lab Background

4.1.1. 목적 및 개요

o 앞선 Lab에서는 Playground를 구성하는 각 박스 내부와 박스 간의 연결 과정을 살펴보았다. Tower Lab에서는 이렇게 준비된 Playground를 관제하기 위하여 Tower를 구축하고, 예제를 통해 Playground의 상태를 확인하는 것을 목표로 한다.

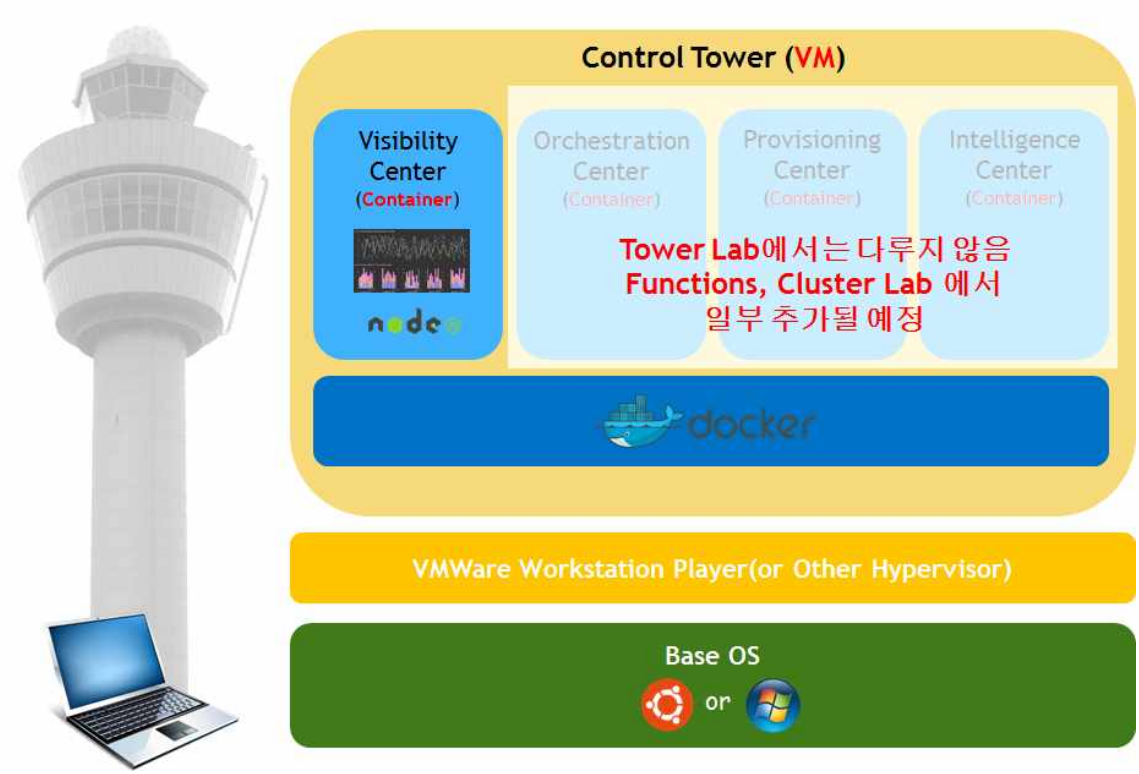


그림 39 Control Tower의 컨셉 및 디자인

o Tower는 개인이 소지한 노트북 등의 PC에 구축한다. 단, 해당 PC는 Playground와 같은 네트워크 상에 있음을 전제로 하며, 윈도우 또는 우분투 리눅스가 설치된 PC에서 편의를 위해 Virtual Machine을 통해 Control Tower를 구축한다. Control Tower 내부는 다시 각각의 기능에 따라 네 가지 Center로 구성되며 Docker 기반의 Container로 작성한다.

o Tower Lab에서는 Control Tower 내의 여러 센터 중에 Visibility Center의 예제로서 각 박스에서 수집된 CPU 상태 정보를 실시간 그래프를 통해 확인한다.

InterConnect Lab에서는 Raspberry Pi가 그 대상이었지만 Tower Lab에서는 NUC도 포함한다.

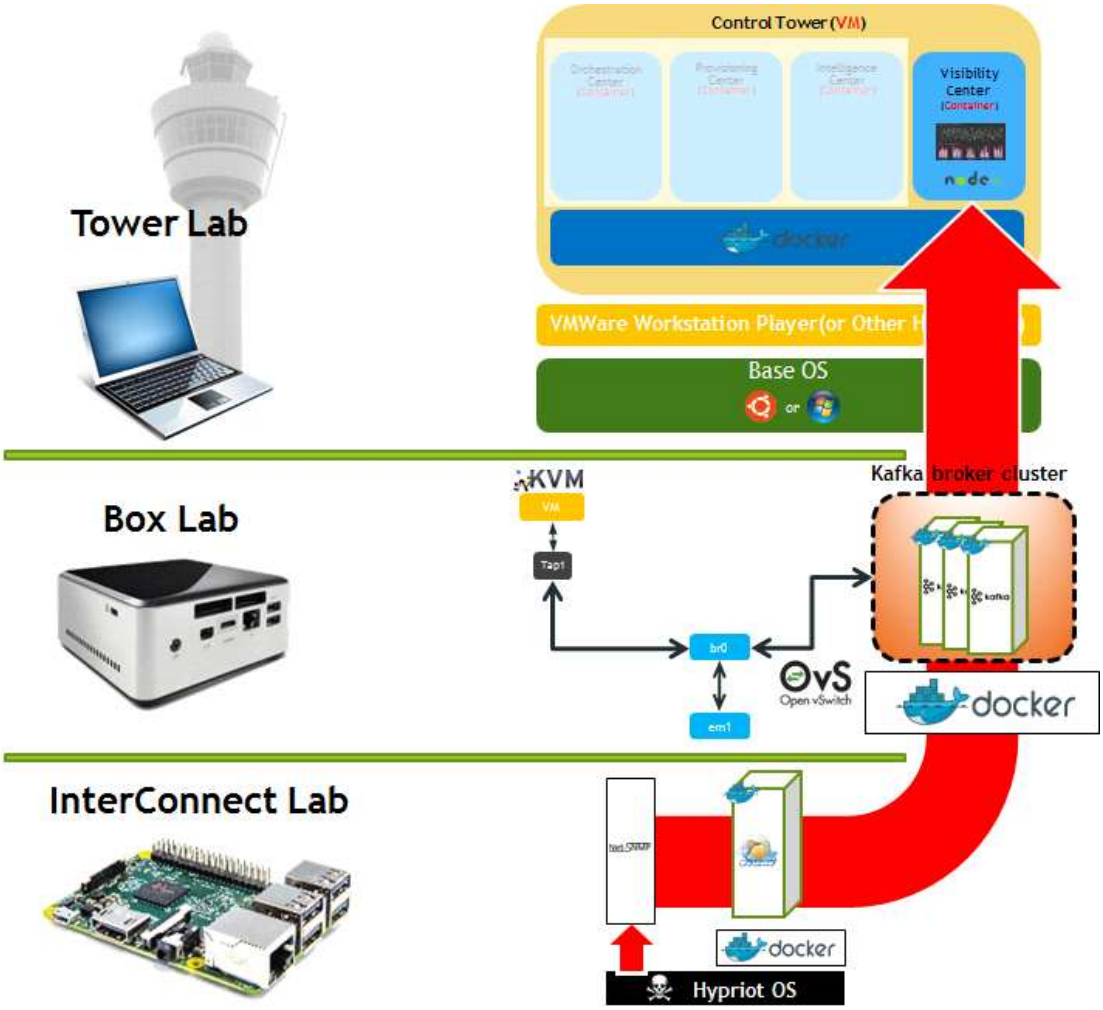


그림 40 Tower Lab과 선행 Lab간의 관계도

o Visibility Center에서는 Kafka Broker에 새로운 메세지가 도착할 경우 즉시 그 메세지를 수집하여 그래프를 통해 시각화 해주는 node.js 기반의 웹 서버를 포함한다. 이를 통해 직접 NUC에 접근할 필요 없이 어떤 상황이 벌어지는지 보다 직관적으로 파악할 수 있다. 예제를 활용한다면 원하는 데이터를 수집하고 이를 적절히 처리하여 결과물을 내놓는 방식의 서비스도 제공할 수 있을 것이다.

o 내용의 순서는 Control Tower와 Visibility Center를 순차적으로 구축한 뒤, Visibility Center의 Operation Data Visibility Service 예제 진행을 위해 NUC의 상태 정보 수집을 위한 Container를 추가로 실행하고 환경에 따른 설정 후 결과 화면을 확인하는 것으로 구성된다.

4.2. Control Tower

4.2.1. Hypervisor 설치

o Hypervisor를 통해 호스트 OS 위에서 여러 게스트 OS가 동작함으로써 격리된 다양한 환경을 다룰 수 있다. VMWare Workstation Player는 VMWare사가 제공하는 여러 솔루션 중 개인 사용자에게 무료로 제공되는 버전이며, 시중에 다양한 무료 하이퍼바이저(VirtualBox 등)가 있으니 상황에 따라 맞춰서 사용하셔도 무관하다.

o 각자 PC 환경에 따라 윈도우 또는 리눅스 버전의 VMWare Workstation Player 설치 파일을 다운 받고 실행시킨 뒤 초기 설정 그대로 설치를 완료한다. 다른 하이퍼바이저를 사용할 경우엔 가상 머신의 네트워크 설정은 NAT로 설정한다.

4.2.2. Virtual Machine 생성

o 설치 완료 후 VMWare Workstation Player를 실행한 뒤, 메인화면에서 Create a New Virtual Machine를 클릭하여 새로운 가상 머신을 생성한다.

o 가상 머신 생성을 위해서는 게스트 OS 설치를 위한 이미지 파일이 필요하므로, 최신버전의 우분투 리눅스 이미지를 다운 받은 뒤 파일의 경로를 지정해준다. Control Tower에서 활용할 계정 정보와 저장 공간의 경로와 그 크기 등을 설정해줌으로써 설치를 마무리한다. 해당 내용은 각자의 상황에 맞춰 적절하게 입력한다.

o 이후 OS 설치 및 계정 생성이 완료되면, Control Tower(VM)에 접속하여 apt-get의 source를 변경합니다. 기본값으로는 우분투 공식 아카이브를 통해 내려받게끔 설정되어 있지만 이따금씩 지나치게 느린 속도로 작업에 문제가 생길 때가 있다.

```
sudo sed -i 's/us.archive.ubuntu.com/ftp.daum.net/g' /etc/apt/sources.list
sudo sed -i 's/kr.archive.ubuntu.com/ftp.daum.net/g' /etc/apt/sources.list
```

4.3. Visibility Center

4.3.1. Docker Engine 설치

o Docker는 어플리케이션을 컨테이너 환경에서 빌드 및 배포, 실행하기 위한 오픈 플랫폼으로, Tower Lab에서는 Docker를 활용하여 Control Center 내의 Visibility Center를 구축한다.

o Docker의 설치 과정은 공식 홈페이지의 가이드에 따르며, 설치 이후 docker 유저 그룹에 사용자를 추가해야 root 권한 없이 docker 명령어를 사용할 수 있다.

```
curl -fsSL https://get.docker.com/ | sh
```

```
sudo usermod -aG docker [username]
sudo service docker restart
```

4.3.2. Docker Image 생성 및 Docker Container 실행

o Github 저장소에서 Docker 이미지 생성에 필요한 파일들을 내려 받는다. Visibility Center를 위한 이미지는 ubuntu-nodejs 폴더 아래에 위치한 Dockerfile 과 플러그인 파일을 필요로 한다.

```
git clone https://github.com/SmartXBox/SmartX-mini.git
cd ~/SmartX-mini/ubuntu-nodejs
```

o 해당 경로에서 Docker 이미지를 빌드한다. 이 과정에서 많은 시간이 소요되므로 사전에 주의가 필요하다.

```
docker build --tag ubuntu-nodejs .
```

o 이미지 빌드가 끝나면 해당 이미지를 통해 컨테이너를 실행시킨다. 이 때 네트워크 설정은 호스트의 네트워크를 그대로 사용한다.

```
docker run -it --net=host --name [container name] ubuntu-nodejs
```

4.4. Use Case: Operation Data Visibility Service

4.4.1. Operation Data 수집을 위한 Container 구동

o 컨테이너 실행에 앞서, Kafka를 통해 메시지를 주고받기 위해서는 IP 주소 기반으로 개체를 식별하는 까닭에 각각의 Kafka broker에 대한 IP 주소 및 호스트 네임을 기록해야 한다. /etc/hosts 파일을 열어 자신의 상황에 맞게 해당 내용을 기입한다.

o Net-SNMP를 통해 Box의 상태 정보를 수집하므로, 이를 위해 snmpd과 MIBs 적용을 위한 다운로더를 받아 적용한다. 이 후 SNMP 쿼리에 대한 응답을 얻기 위해 localhost에 대한 보안 설정 항목을 수정하고 snmpd을 재시작한다.

```
sudo apt-get update
apt-get install –y snmp snmpd snmp-mibs-downloader
download-mibs
vi /etc/snmp/snmpd.conf
(#rocommunity public localhost -> Delete #)
/etc/init.d/snmpd restart
```

o Flume이 동작하기 위한 이미지를 빌드 한 뒤, 컨테이너를 실행한다. 앞서 Github 에서 다운 받은 경로 내의 ubuntu-flume 아래에 해당 Dockerfile이 위치하며, 앞선 과정과 마찬가지로 긴 시간이 소요될 수 있음에 유의한다.

```
cd SmartX-mini/ubuntu-flume
docker build --tag ubuntu-flume .
docker run –it --net=host --name ubuntu-flume ubuntu-flume
```

o 컨테이너가 실행되면 localhost에서 수집한 데이터를 Kafka brokers로 전달하기 위해 설정 파일의 수정이 필요하다. 앞서 hosts 파일을 수정했던 내용을 바탕으로 conf/flume-conf.properties 파일 내에서 토픽의 이름과 브로커 리스트를 수정한다.

```
vi conf/flume-conf.properties
agent.sinks.sink1.topic=[topic_name]
agent.sinks.sink1.brokerList=[broker_ipaddress:port]
```

o 모든 설정을 마친 뒤, Flume agent를 실행하여 상태 정보 전달을 시작한다. 전달된 정보는 Kafka broker에 임의의 Consumer를 추가하여 확인 가능하며, 자세한 내용은 앞선 InterConnect Lab을 참조한다.

```
bin/flume-ng agent --conf conf --conf-file conf/flume-conf.properties --name agent -Dflume.root.logger=INFO,console
```

4.4.2. NUC 설정 Visibility Center 환경 설정

o Visibility Center 내의 node.js 파일을 열어 모니터링 대상의 IP 주소와, 해당 정보가 수집되는 Kafka Broker에 대한 설정을 수정한다.

```
vi node.js
```

o 그림 3의 경우 1개의 Raspberry Pi와 1개의 NUC에 대하여 운영 상태 정보를 수집하며, 해당 내용은 master1이라는 호스트 네임을 가진 박스에서 2181포트를 통해 통신하는 Kafka broker를 통해 전달된다. 이 때 토픽의 네임은 0506_3이고, 해당 토픽은 3개의 파티션으로 분할되어 있음을 확인할 수 있다.

o 모든 설정을 반영 한 뒤 node.js 파일을 실행하여 상태 정보에 대한 Visualization 을 시작한다

```
node node.js
```

```
var rpi = '192.168.88.101',
nuc = '192.168.88.147';

var flagRpi,flagNUC =0;

var express = require('express'),
 fs = require('fs'),

 kafka = require('kafka-node'),
    Consumer = kafka.Consumer,
    client = new kafka.Client('master1:2181'),
    topics = [{topic: '0506_3', partition:0},
{topic:'0506_3',partition:1},
{topic:'0506_3',partition:2}];
```

그림 41 node.js 파일 내에서 상태 정보 수집을 위한 설정 변경의 예

4.4.3. Operation Data Visibility Service 실행

o node.js 파일을 실행 뒤 웹 브라우저를 통해 아래 주소로 접속하면 실시간으로 변하는 상태 정보에 대한 차트를 확인할 수 있다.

http://localhost:3000

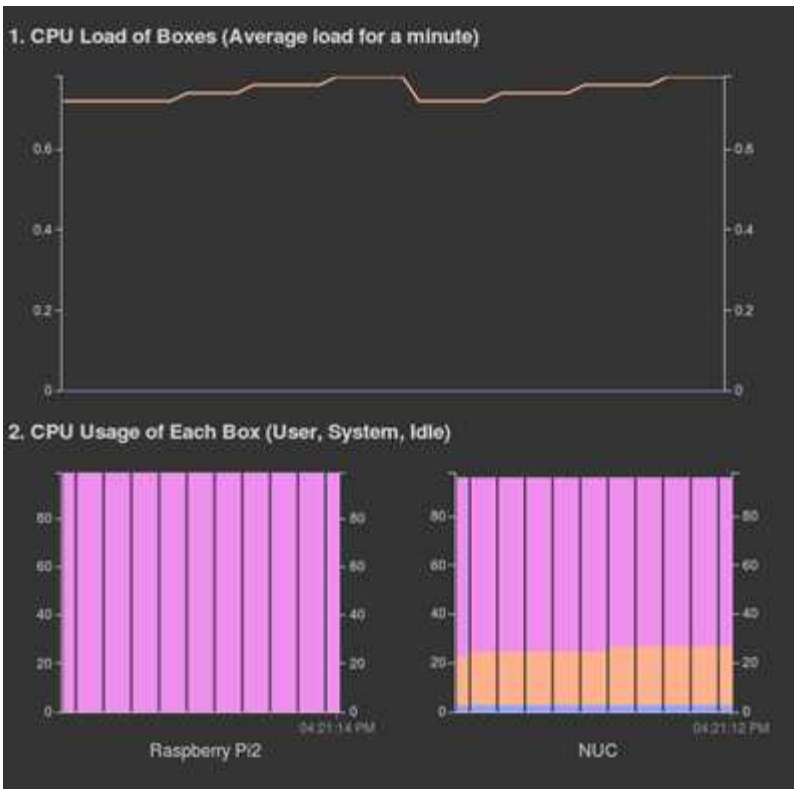


그림 42 각 Box별 Load 비교 및 Box내의 CPU 사용 현황

4.5. 결론

o 본 기술문서의 Tower Lab은 전체 Playground를 관제하기 위한 Control Tower의 개념을 이해하고, 다양한 오픈 소스를 활용하여 Tower를 구성하는 일부 Center를 직접 구현하게끔 구성하였다.

o 이에 따라 향후 IoT-Cloud 환경을 위한 Plagyround 구성에 대한 이해를 돕고, Docker 기반의 컨테이너 기술과 Kafka를 통한 메시징 시스템을 활용한 다양한 서비스 개발에 도움이 될 것으로 기대한다.

5. Functions Lab

5.1. Functions Lab Background

5.1.1. 목적 및 개요

o Functions Lab은 지금껏 구성된 SmartX-Mini Playground에서 Container를 이용해 Microservice를 이해하고 SmartX-Mini Playground의 Function을 전반적으로 다뤄보는 과정을 담고 있다. Docker를 통해 직접 Container들을 만들고 그 안에 Docker Repository에서 Functions들을 배치하며 Microservice를 이해하는 것에 초점을 둔다.

o 본 기술문서의 Function Part의 구성환경은 다음과 같다. Ubuntu Linux가 설치된 하나의 물리적인 머신(Intel NUC)에서 Lab을 진행한다. 그리고 Ubuntu Linux 환경에서 docker를 사용해 Container를 추가/제거하고 거기에 Function들을 배치하는 것을 다뤄본다.

5.1.2. Function 및 Microservice

o Microservice는 기존의 Monolithic 서비스 구조의 단점을 개선하기 위해 나온 방식이다. 먼저 Monolithic 서비스 구조에 대해서 말하자면, Monolithic 서비스는 전체 서비스를 하나의 거대한 프로그램이 제공하는 형식으로, 이 거대한 프로그램 하나가 서비스의 각 모든 기능들을 관장한다. 일견 보기에 이 구조는 간단해 보일 수 있으나, 점점 서비스의 규모가 커지고 서비스의 유지보수 및 개발에 참여하는 인원이 늘어나면 단점이 두드러지기 시작한다.

Monolithic 구조의 서비스는 시간이 지나 그 서비스의 규모가 너무 커지면 Application의 크기도 덩달아 커져 도중 후속 개발자 등이 세부 내용을 완전히 이해하거나 변경을 빠르고 정확하게 하기가 점점 힘들어진다. 기능들 중 하나만 오류가 발생해도 전체 시스템이 마비될 수 있으며, 서비스의 일부 수정만으로도 이를 적용하기 위해서는 Application 전체를 재배포하고 전체 서비스를 멈춰야 하는 문제가 있다. 또한 특정 기술에 대한 종속이 심해질 수 있다. 예를 들어 JVM 기반으로 작성된 Monolithic Application에 기반한 서비스는 JVM 기반을 탈피하기 위해서는 한꺼번에 전체 Application을 재작성하지 않는 이상 JVM 종속성에서 결코 벗어날 수 없다.

이에 대응하는 구조가 Microservice 구조로, 이 구조에서는 전체 서비스를 각 기능 단위의 작은 Application들로 쪼개 따로 배치한 후 그 Application들을 엮어 하나의 전체 서비스를 구성한다. 여러 작은 기능들이 모여서 돌아가고 있으므로, 만일 서비스의 특정 부분을 변경했다면 그 서비스를 담당하고 있는 Application만 변경하고 그 기능 부분만 재시작하면 되고, 한 기능의 오류가 발생해도 전체 서비스 중 그 기능에 의존하고 있는 부분을 제외한 부분들은 영향받지 않는다. 또한

기능별 단위로 쪼개져있으므로 분산 시스템에 더욱 적합하며, 각각의 기능들은 따로 배치되고 따로 개발될 수 있다. 그리고 각각의 기능별로 쪼개져 있으므로 특정 기술에 대한 종속성이 훨씬 감쇄된다. 하나의 기능은 JVM 기반이면서 다른 기능은 네이티브 기반으로 작성하는 것도 당연히 가능하다.

o Microservice에서의 잘게 쪼개진 기능을 Function이라고 칭한다. 여기서의 Function은 그러한 기능을 담당하는 Software를 추상화된 개념으로, 앞선 Lab에서 언급된, 하드웨어를 추상화해 개별적인 하드웨어의 속성을 분리해낸 개념인 Box에 대응된다. 순수한 소프트웨어 측면에서 이러한 Function들이 사슬처럼 얽히고 모여 Service chain을 이루고, 이렇게 모인 Function들이 전체 서비스를 구성한다.

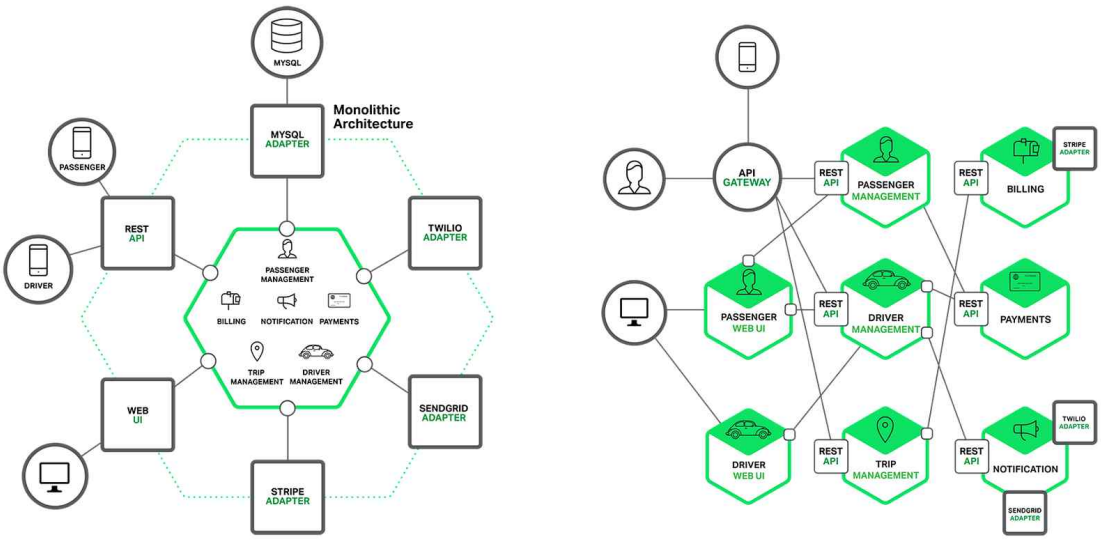


그림 43 Monolithic Application(좌) 및 Microservice(우)

5.1.3. Container

o Container는 하나의 커널을 공유하는 여러 개의 격리된 Userspace 인스턴스 형태인 Operation System Level Virtualization에서 각각의 격리된 사용자 공간들을 칭한다. Baremetal, Hosted 가상화가 하나의 Hypervisor 위에 각각의 격리된 인스턴스인 Virtual Machine들이 놓이고 그 안에 각각의 Kernel들이 또 배치되는 것과 대조를 이룬다. Container는 이러한 점에서 자신만의 커널이 없이 호스트 운영체제의 커널을 공유한다는 점에 차이점을 이룬다. 이러한 특징을 통해 Container는 앞서 언급한 Full Virtualization인 Baremetal, Hosted 구조에 비해 훨씬 가볍고 빠르다. 다만 그 대신 자신만의 커널을 설치할 수 없기 때문에 반드시 호스트 운영체제와 같은 OS를 이용해야 하는 단점이 있다.

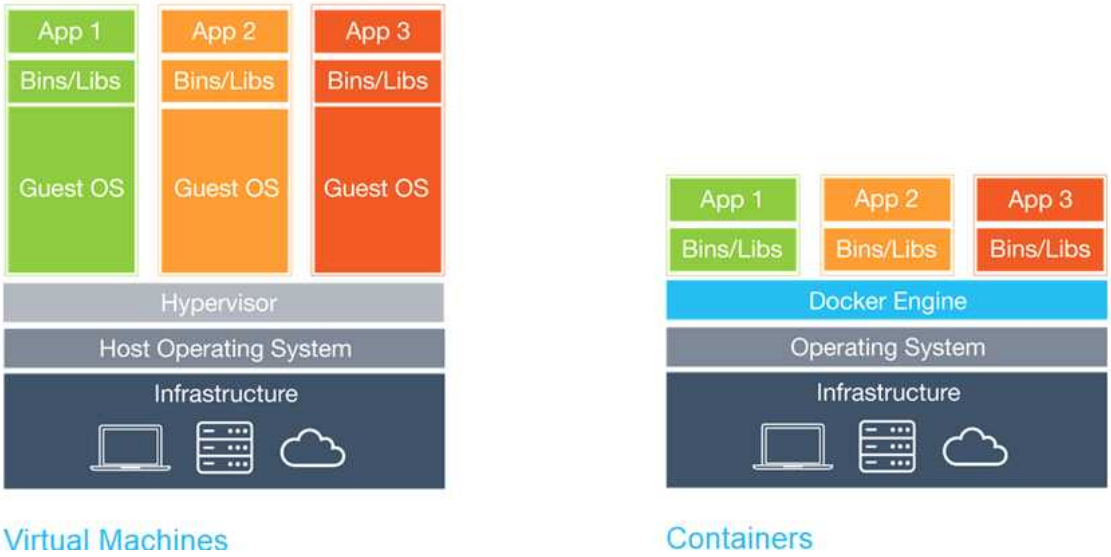


그림 44 Virtual Machine과 Container의 차이

5.1.4. Docker

o Docker는 리눅스 Application들을 Container에 배치하는 것을 자동화시켜주는 오픈 소스 프로젝트이다. Docker는 cgroup, 커널 namespace 등의 리눅스 커널의 자원 분리 기능을 활용해 각각의 Container들을 호스트 OS로부터 분리해낸다. 이러한 기능을 통해 각각의 Application들은 OS의 관점으로부터 거의 완벽하게 분리되어 각각의 독립된 환경에서 구동될 수 있게 된다. 또한 Docker는 git과 유사한 이미지 저장소를 가지며, 개인 저장소를 쓸 수도 있고, 공개된 저장소에 접근해 다른 사람 및 단체가 배포하는 이미지를 받아 Application들을 손쉽게 분산된 시스템에 배치하거나, 역으로 배포할 수도 있다. 이러한 Docker 이미지들은 각각의 이미지를 재현하는데 필요한 순차적인 내용이 담긴 Dockerfile이라는 대체로 작은 텍스트 파일 형태로 그 기록이 저장되어 커다란 용량이 필요하지 않다. hub.docker.com에서는 여러 단체 및 사용자들이 만들고 공개한 수많은 Docker 이미지들이 배포 & 관리되고 있다.

5.2. Settings & Configuration

5.2.1. 예제 구성

o 이 기술 문서에서는 예시로 Nginx, Wordpress, MySQL로 구성된 3-tier Function Chain을 구성한다. 이 서비스는 Web Function인 Nginx를 프록시 서버로서 운용하며 여기에 TCP 80 포트에 HTTP로 /black에 접근 시 가칭 Wordpress “Black”의 TCP 8888 포트로, /red에 접근 시 Wordpress “Red”의 TCP 9999 포트로 리다이렉트되어 각각의 웹페이지에 연결된다. 그리고 각각의 Wordpress App Function들은 각자 독립된 MySQL DB Function들과 연결된다.

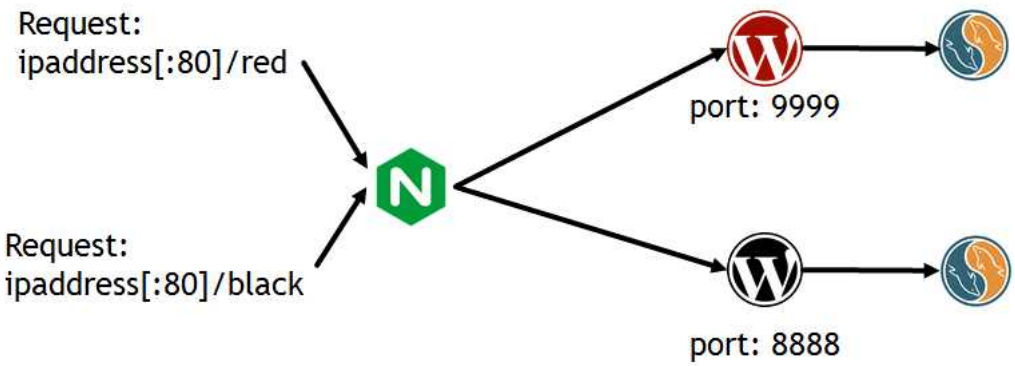


그림 45 예제 3-Tier Function Chain

o 또한 이 기술문서에서는 Ubuntu Linux 14.04, docker-engine 1.10 혹은 그 이상 버전을 기준으로 설명하고 있다. 또한 docker 사용자가 sudo 그룹에 소속되어 있어 docker 명령어를 슈퍼유저 권한 없이 사용할 수 있게 된 상태를 기준으로 잡고 있다.

5.2.2. 설정 및 구성

o MySQL은 아래의 명령어로 간단하게 설치할 수 있다. 아래 명령어에서 빨간색은 사용자가 임의로 변경할 수 있는 부분으로, --name 옵션 뒤의 word_sql은 생성할 Container의 이름, 그리고 -v 옵션 뒤의 [username]에는 OS에서의 사용자의 ID, [password]에는 생성한 Container에 담길 MySQL DB Function에서 DB의 root 사용자의 비밀번호를 결정한다. 또한 -d 옵션 뒤의 5.7.12는 저장소에 있는 mysql 이미지의 버전 브랜치를 명시한 것이다. 생략되면 자동으로 가장 최신버전을 의미하는 latest 브랜치를 택해 이미지를 받게 된다. 사용자의 로컬 저장소에 만일 mysql 이미지가 없다면 자동으로 hub.docker.com에서 명시된 이미지를 다운로드 받아 설치하게 된다.

```
mkdir ~/sql
docker run --name word_sql -v /home/[username]/sql:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=[password] -d mysql:5.7.12
```

o 위의 명령어는 word_sql이란 이름의 Container를 하나 만들고, 호스트 OS에서의 디렉토리 /home/[username]/sql를 Container 볼륨 내의 /var/lib/mysql과 하드 링크시킨다. 그리고 이미지로는 mysql 이미지의 5.7.12 브랜치를 사용하며 설치 시에 root 계정의 암호는 [password]로 지정한다.

o 앞서 MySQL을 설치했던 것과 마찬가지로 WordPress App Function도 설치한다.

```
docker run --name wordpress --link word_sql:mysql -p 8888:80 -d
wordpress:4.5.1-apache
```

o 위의 명령어에서 사용된 –link 옵션은 word_sql을 새로 생성한 wordpress Container가 접근할 수 있게 해주며, Alias를 mysql로 설정한다. -p 옵션은 Container word_sql를 외부에서 제한적으로 접근할 수 있게 하는 것으로, 호스트의 8888번 포트와 word_sql의 80번 포트와 1:1 연결을 해준다. 즉 호스트의 8888번 포트에 접속하면 word_sql의 80번 포트로 리다이렉트되게 되는 것이다. 그리고 워드프레스는 기본적으로 웹 서버에서 구동되는 웹앱이므로, 컨테이너에 워드프레스를 구동시켜 줄 웹서버 또한 필요한데, 여기서 사용되는 wordpress:4.5.1-apache는 Apache 웹서버를 이용한다.

o 만들어진 Container는 아래와 같이 docker ps 명령어를 통해 확인할 수 있다.

```
tein@vbox-develop:~$ docker ps
CONTAINER ID        IMAGE                    COMMAND                  CRE
ATED              STATUS              PORTS                    NAMES
752dcdfdd388        wordpress:4.5.1-apache   "/entrypoint.sh apach"   3 s
econds ago        Up 2 seconds        0.0.0.0:8888->80/tcp     wordpress
5cba5d67f49c        mysql:5.7.12             "docker-entrypoint.sh"   5 m
inutes ago        Up 5 minutes        3306/tcp                 word_sql
```

그림 46 만들어진 컨테이너의 확인

o 그리고 웹 브라우저를 열어 localhost:8888로 접속한 뒤에 워드프레스 웹앱을 설정한다. 그리고 사이트 제목을 black으로 설정한다. black 외의 다른 제목을 설정해도 좋지만, 아래에서 따로 추가할 새로운 워드프레스 웹앱과는 다른 이름을 두어 구분할 수 있게 해야 한다. 이렇게 되면 워드프레스 “Black”의 설정이 끝난다.

그림 47 Wordpress “Black” 설정

o 그리고 그 다음으로는 워드프레스 “Red”를 설정한다. “Black” 설정과 일부 값만 다를 뿐, 사용하는 명령어는 같다.

```
mkdir ~/sql2
docker run --name word_sql2 -v /home/[username]/sql2:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=[password] -d mysql:5.7.12
docker run --name wordpress2 --link word_sql:mysql -p 9999:80 -d wordpress:4.5.1-apache
```

o 그 후에는 웹 브라우저에서 localhost:9999에 접속해서 워드프레스 “Red”의 웹앱을 설정한다. 이번 사이트는 제목을 red로 설정한다. 물론 “Black” 설정 때와 같이 사용자 마음대로 변경해도 문제없다.

그림 48 Wordpress “Red” 설정

o 두 워드프레스 Function의 설정이 완료되었다면, 이제 Nginx Function을 구성한다. 그러나 Nginx는 워드프레스나 MySQL처럼 제대로 된 저장소가 없으므로 Ubuntu 컨테이너를 하나 만든 뒤 거기에 Nginx를 직접 설치한다. 아래의 명령어는 nginx라는 이름의 컨테이너를 만들고 그 안에 ubuntu 14.04 환경을 구성한다.

```
docker run -it -name=nginx -p 80:80 ubuntu:14.04
```

o 위의 설정은 호스트에 80번 포트로 접속하게되면 nginx 컨테이너의 80번 포트로 리다이렉트되게 만든다. HTTP 기본 포트가 80번인 것을 감안한다면 호스트에 웹 브라우저로 접속하게 되면 nginx 컨테이너로 바로 연결하게 되는 것이다.

o 그리고 nginx 컨테이너의 구성이 끝나면 nginx 컨테이너에서 bash 셸로 들어간 뒤 아래의 명령어들을 입력해 nginx와 vim을 설치한 뒤 vim으로 Nginx의 설정 파일을 열고, 그림 7과 같은 내용들을 써넣는다. 단, 그림에서 http://뒤의 빈 자리에는 호스트의 IP를 기록한다.

```
apt-get update
apt-get install nginx
apt-get install vim

cd /etc/nginx/sites-enabled
rm default
vi default
```

```
server {
        listen 80;
        location /black {
                proxy_pass http://              :8888/;
        }
        location /red {
                proxy_pass http://              :9999/;
        }
}
```

그림 49 Nginx의 설정

o 위의 설정에서 listen 80는 80번 포트로 접속할 수 있게 하고, location /black은 nginx에 /black으로 접속했을 때의 설정을 다룬다. 그리고 여기서 proxy_pass 명령을 통해 /black으로 접속했을 때 호스트의 8888번 포트의 /로 리다이렉트시켜 주게 된다. 그리고 앞서 호스트의 8888번 포트는 wordpress의 8888번 포트와 1:1 매핑된 상태이므로 wordpress 컨테이너로 접속하게 된다. location /red 또한 비슷하게 동작해 호스트의 9999번 포트의 /를 통해 wordpress2 컨테이너로 접속하게 된다.

o 그리고 설정이 끝나면 아래의 명령어를 입력해 Nginx를 실행시킨다.

```
service nginx start
```

o 이렇게 되면 예제의 Function Chain의 구성이 끝나며 웹 브라우저를 통해 http://localhost:black, http://localhost:red로 접속 시 아래와 같은 결과가 나타나게 된다.

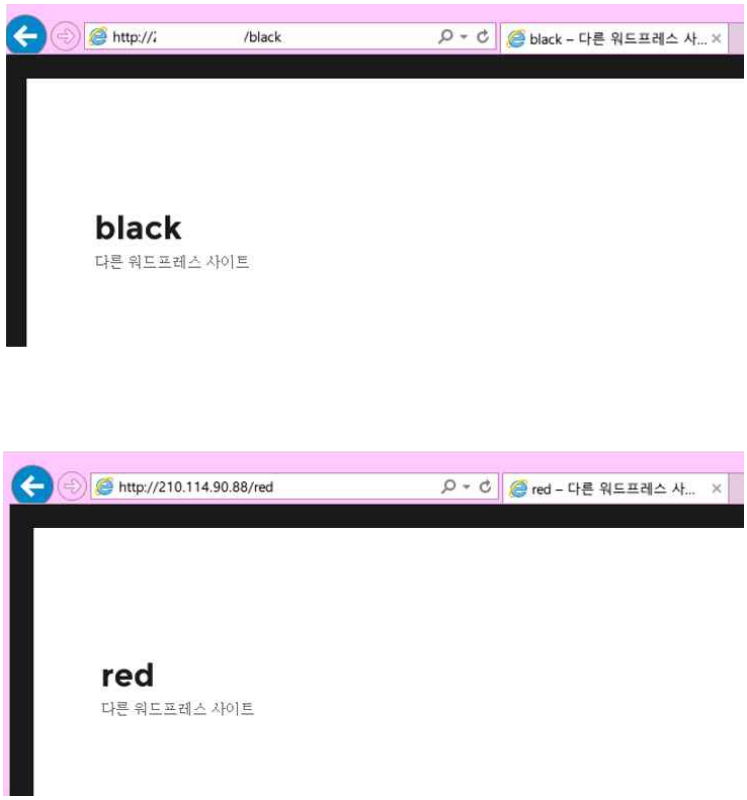


그림 50 최종 결과

5.3. 결론

o 본 기술문서 Functions part는 Function 개념이 생소한 사람들을 대상으로 작성되었으며 Function의 개념을 비롯해 Docker의 구성 방식과 간단한 사용법을 배워보는 식으로 구성이 된다.

6. WebApp Lab

6.1. WebApp Lab Background

6.1.1. 목적 및 개요

o WebApp Lab은 node.js 기반의 웹 애플리케이션을 사용자가 직접 제작하고, Raspberry Pi의 센서를 활용한 프로그램을 구현하는 Lab이다. 지금껏 구성된 SmartX Playground를 기반으로 하는 IoT-Cloud 서비스의 실증을 목표로 한다.

o 본 기술문서의 WebApp Part의 구성환경은 다음과 같다. 앞선 Lab에서 Intel NUC에 설치한 Docker 기반의 컨테이너 내부에 node.js 기반의 WebApp을 통해 Raspberry Pi – NUC – Tower의 3 tier 구조 웹 애플리케이션을 구축하는 과정을 담고 있다.

6.1.2. node.js

o node.js는 구글의 크롬 V8 자바스크립트 엔진을 기반으로 하는 네트워크 서버 프레임워크이다. 자바스크립트 기반으로 프로그램의 작성이 이루어 지며, 싱글 스레드 기반의 이벤트 루프를 특징으로 하며 비동기 IO 처리를 통해 빠른 성능을 보인다는 장점을 가진다. 아래 그림은 node.js의 구조를 나타낸다.

그림 51 node.js 웹 애플리케이션의 구조

6.2. Settings & Configuration

6.2.1. 예제 구성

그림 52 WebApp Lab 3 tier 구조

o 이 기술 문서의 WebApp Lab 파트의 세부 내용은 node.js 개발 환경 및 기본적인 예제 코드, Raspberry Pi 온습도 측정 앱 구축, node.js 기반 WebApp 구축의 세 단계로 구성된다.

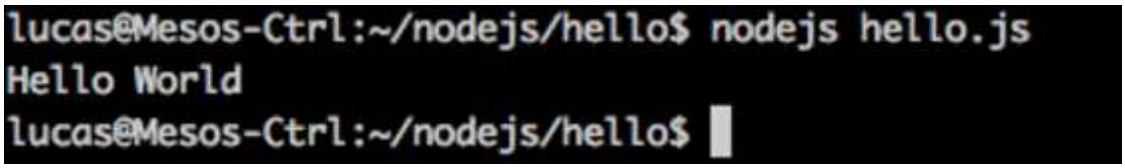
6.2.2. 설정 및 구성

o node.js 개발 환경의 구축을 위해 우선 Intel NUC에 node.js 패키지를 설치한다. 설치 방법은 아래와 같다.

```
$ sudo apt-get update
$ sudo apt-get install nodejs npm
```

o 위의 명령어를 통해 node.js의 설치를 완료한 후 작동 확인을 위해 아래와 같은 예시 코드를 작성한다. node.js가 정상적으로 설치된 경우 아래 그림과 같이 터미널 창에 Hello World라는 결과가 나오게 된다.

```
$vim hello.js
console.log('Hello World')
```



o node.js를 설치한 후에는 node.js에서 사용할 수 있는 모듈들을 관리해주는 node package module을 설치한다.

```
$sudo apt-get install npm
```

o Docker Container 내부에 node.js 기반 WebApp을 구동하기 위해 아래 명령어를 통해 컨테이너 생성 및 기본 설정을 해준다.

```
$sudo docker run -it —net=host —name=webapp ubuntu /bin/bash
# apt-get update
# apt-get install net-tools
# apt-get install iputils-ping
# apt-get install nodejs
```

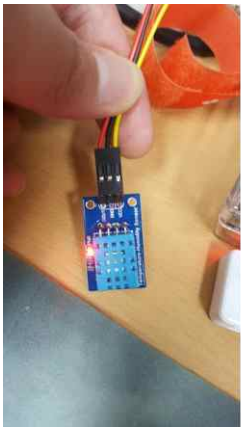
o 이후 텍스트 에디터를 통해 아래와 같은 WebApp의 코드를 작성한다.

```
# nano webapp.js

var http = require('http');
var url  = require('url');
var fs = require('fs');
var temp="";

http.createServer(function  (request, response) {

   var query = url.parse(request.url, true).query

   response.writeHead(200, { 'Content-Type': 'text/html' });
   console.log(JSON.stringify(query));



   if(JSON.stringify(query).length>13)
{
   fs.writeFile('temp.txt', JSON.stringify(query), 'utf8', function (error){
   console.log('write');
});
}
   fs.readFile('temp.txt', 'utf8', function (error, data) {
console.log(data);
temp = data;
});
response.end(temp);

}).listen(80, function () {
   console.log('Server running...');
});
```

o 작성한 WebApp을 컨테이너 내부에서 실행시킨 이후 NUC의 웹 브라우저를 통해 http://<ip of nuc> 에 접속해 WebApp의 구동 여부를 확인한다.

o IoT 파트의 온도센서 프로그램을 구현하기 위해 Raspberry Pi와 온도 센서를 다

음 그림과 같이 연결한다.



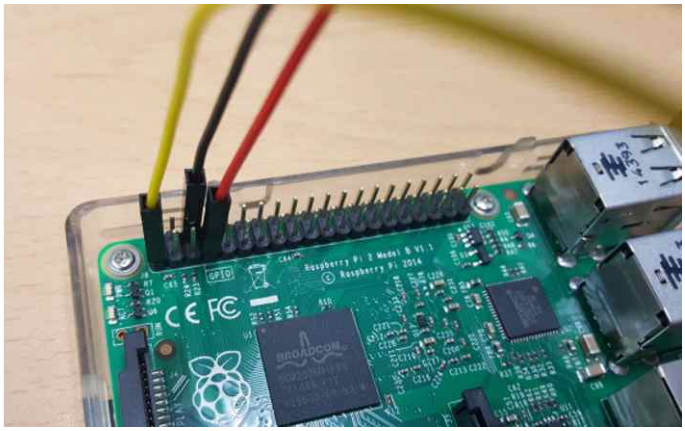


그림 54

o Raspberry Pi에 온도센서를 연결한 후 온도센서가 정상적으로 작동하기 위한 패키지들을 아래 명령어를 통해 설치한다.

```
$ sudo apt-get update
$ sudo apt-get install python-pip
$ sudo apt-get install libpython2.7-dev python-numpy
$ sudo pip install RPi.GPIO
$ sudo apt-get install mercurial
```

o 모든 패키지를 설치한 후 온습도를 측정해 node.js 기반 웹 애플리케이션으로 정보를 전송하는 프로그램을 아래와 같이 작성한다.

```
$ sudo nano RPI_temp.py

import RPi.GPIO as GPIO
import time
import urllib2

def bin2dec(string_num):
    return str(int(string_num, 2))


data = []

GPIO.setmode(GPIO.BCM)

GPIO.setup(4,GPIO.OUT)
GPIO.output(4,GPIO.HIGH)
time.sleep(0.025)
GPIO.output(4,GPIO.LOW)
time.sleep(0.02)

GPIO.setup(4, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

```
while GPIO.input(4) == 0:
        pass
while GPIO.input(4) == 1:
        pass
for i in range(0,2000):
    data.append(GPIO.input(4))

#print data
bit_count = 0
tmp = 0
count = 0
HumidityBit = ""
TemperatureBit = ""
crc = ""



try:
        while data[count] == 1:
                tmp = 1
                count = count + 1

        for i in range(0, 32):
                bit_count = 0

                while data[count] == 0:
                        tmp = 1
                        count = count + 1

                while data[count] == 1:
                        bit_count = bit_count + 1
                        count = count + 1

        if bit_count > 9:
                        if i>=0 and i<8:
                                HumidityBit = HumidityBit + "1"
                        if i>=8 and i<16:
                                pass
```

```
if i>=16 and i<24:
                        TemperatureBit = TemperatureBit + "1"
                if i>=24 and i<32:
                        pass
        else:
                if i>=0 and i<8:
                        HumidityBit = HumidityBit + "0"
                if i>=8 and i<16:
                        pass
                if i>=16 and i<24:
                        TemperatureBit = TemperatureBit + "0"
                if i>=24 and i<32:
                        pass
#       print "Data : "
#       print "Humidity : " + HumidityBit
#       print "Temperature : " + TemperatureBit
#       print da1
#       print da2

except:
        print "ERR_RANGE 1"
#       exit(0)

try:
        for i in range(0, 8):
                bit_count = 0

                while data[count] == 0:
                        tmp = 1
                        count = count + 1

                while data[count] == 1:
                        bit_count = bit_count + 1
                        count = count + 1

        if bit_count > 12:
                        crc = crc + "1"
```

```
                else:
                                crc = crc + "0"
except:
        print "ERR_RANGE 2"
#       exit(0)

Humidity = bin2dec(HumidityBit)
Temperature = bin2dec(TemperatureBit)

if int(Humidity) + int(Temperature) - int(bin2dec(crc)) == 0:
        print "Humidity:"+ Humidity +"%"
        print "Temperature:"+ Temperature +"C"

urllib2.urlopen("http://192.168.88.85?temp="+Temperature+"humid="+Humid
ity).close
else:
        print "Result"
        print Humidity
        print Temperature
#       print "ERR_CRC"
```

o 이후 Raspberry Pi에서 위 프로그램을 구동하면 아래와 같은 결과를 확인할 수 있다. 또한 NUC에서 구동중인 WebApp에서도 온습도가 표시되는 모습을 확인할 수 있다.

그림 55 온습도 정보

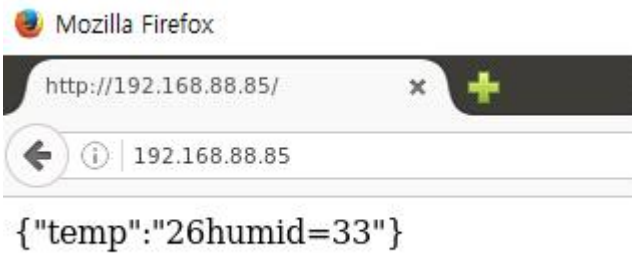


그림 56 WebApp 구동 결과

6.3. 결론

o 본 기술문서의 WebApp Lab 부분을 통해 간단한 node.js 기반 웹 애플리케이션을 구현하고, Raspberry Pi와 연동하는 프로그램을 작성하였다. 본 Lab의 예제 코드들을 활용하여 원하는 IoT-Cloud 서비스를 구현할 수 있다.

7. Cluster Lab 개요

7.1. Cluster Lab Background

7.1.1. 목적 및 개요

o Cluster Lab은 빅데이터 처리를 체험해보는 Analytics Lab에 앞서 진행해야 하는 과정으로, Analytics Lab에서 사용할 환경을 준비하기 위해 관련 소프트웨어들의 설치 과정을 실습하는 내용으로 구성되어 있다. 이 실습을 통해 클라우드에서의 데이터 분산 처리 환경에 대해 이해하고 여기에 어떤 소프트웨어들이 사용되는지 알아보는 것을 목표로 하며, 다음 실습인 Analytics Lab이 문제없이 진행될 수 있도록 한다.

o 본 기술문서의 NUC Cluster의 구성환경은 NUC 1대를 Tower로, NUC 2대를 작업 노드로 사용하여 Tower NUC에는 Mesos Master, Spark, Zeppelin을, 다른 NUC 2대에는 Mesos Slave를 설치한다.

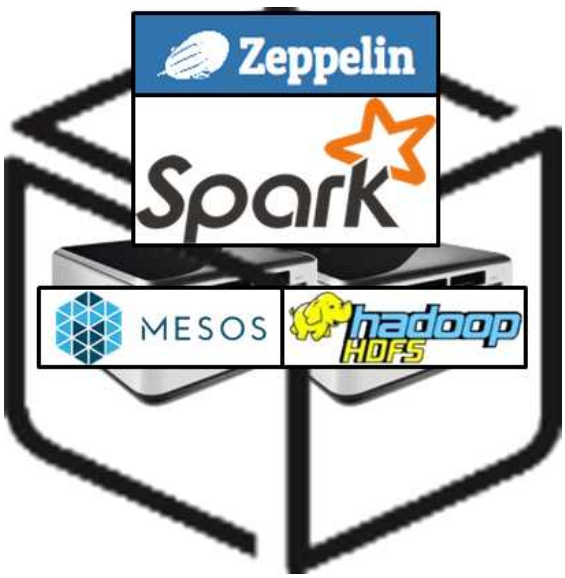


그림 57 Cluster Lab Overview

7.2. Softwares

7.2.1. Apache Mesos

o Apache Mesos는 오픈소스 클러스터 관리 소프트웨어로, Hadoop이나 Spark 같은 분산 환경에서 동작하는 프레임워크들에 대해 클러스터의 자원을 동적으로 할당해주며 이를 통해 분산 자원의 효율적인 운영이 가능해지고 여러 대의 기기로 구성된 클러스터나 데이터센터 전체를 마치 하나의 기기처럼 사용할 수 있게 해

준다.

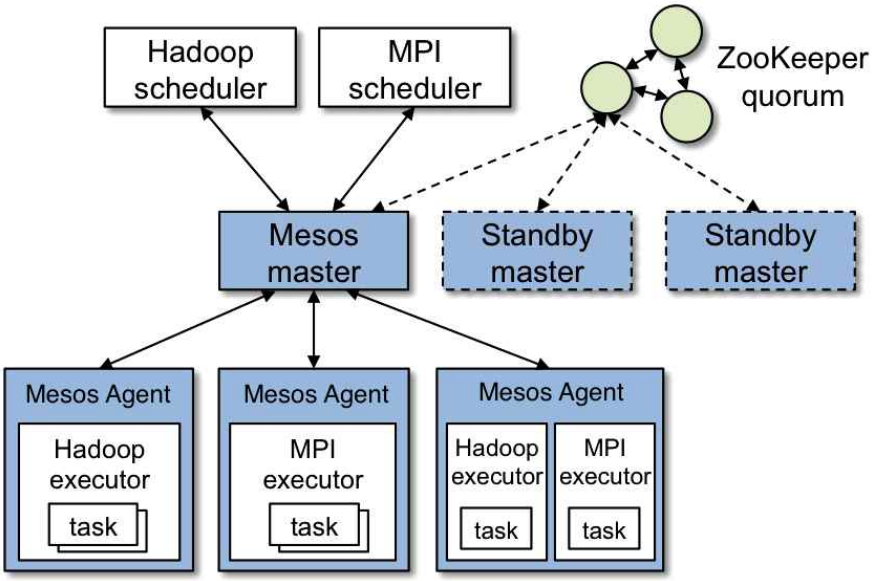


그림 58 Mesos의 구조

7.2.2. HDFS(Hadoop Distributed File System)

o HDFS(Hadoop Distributed File System)은 Apache Hadoop 프로젝트에서 분산 환경에 파일을 저장하기 위한 분산 파일 시스템이다. Hadoop이 사실상의 빅데이터 플랫폼의 표준으로 자리잡은 시점에서 HDFS는 가장 널리 사용되고 있는 분산 파일 시스템 중 하나이다. HDFS는 확장성, 내고장성, 안정성 등의 특징을 가지고 있다.

o HDFS는 파일 시스템에 저장된 파일들의 메타데이터를 관리하는 NameNode와 실제로 파일이 저장되는 DataNode로 구성된다. 또한 NameNode에 장애가 발생했을 때를 대비하여 Secondary NameNode가 존재한다. HDFS에서 파일에 접근할 때는 NameNode를 통한다.

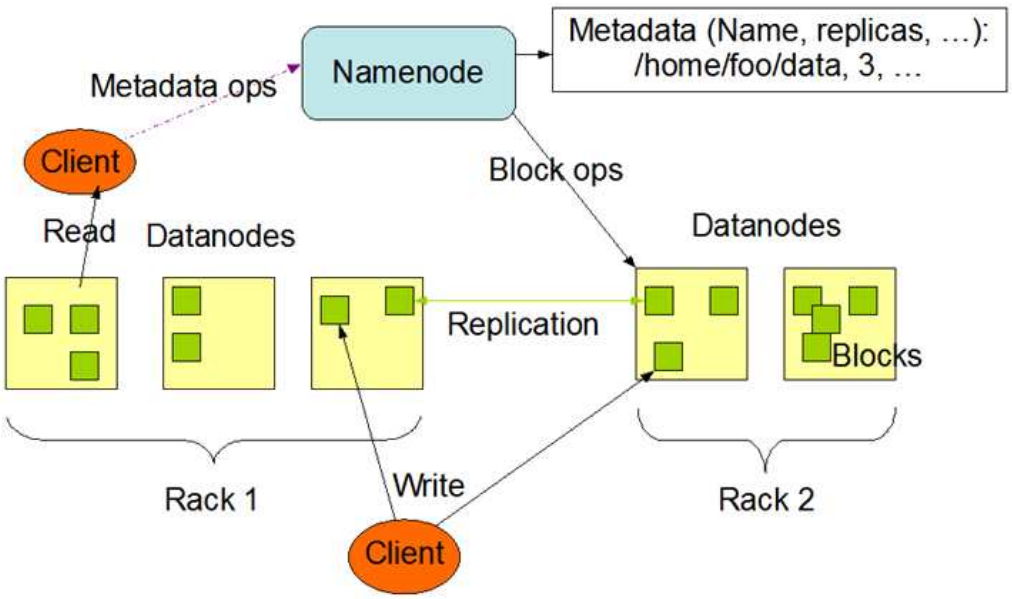


그림 59 HDFS 아키텍처의 작동 원리

7.2.3. Apache Spark

o Apache Spark는 대규모 데이터 처리를 위한 오픈소스 기반의 범용 분산 컴퓨팅 플랫폼으로, RDD(Resilient Distributed Dataset)를 이용한 인메모리 컴퓨팅을 제공하여 기존의 하드디스크 기반 Map&Reduce에 비해 훨씬 빠른 속도를 제공한다. Spark는 Scala, Java, Python, R을 지원하며, Spark Streaming, SparkSQL, SparkR 라이브러리와 다양한 서드파티 패키지를 지원하여 스트리밍 처리, SQL 쿼리, 머신 러닝 등 과학 계산을 위한 R 함수들을 사용할 수 있다.

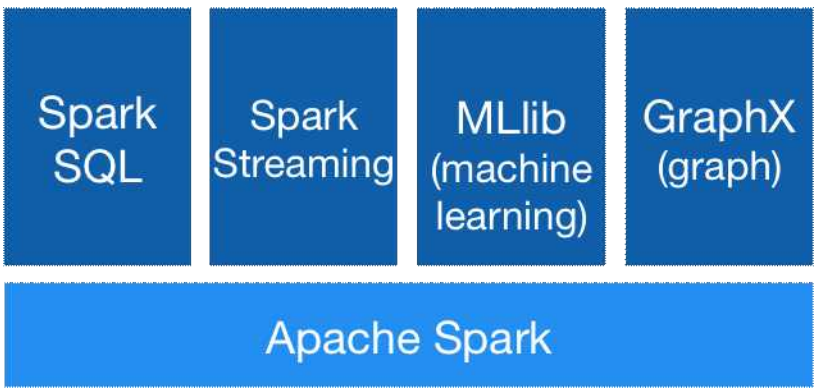


그림 60 Apache Spark의 프레임워크

o Spark는 다음과 같은 점에서 Analytics Engine의 주요 역할을 담당하게 된다.

- Dataset 및 DataFrame과 SparkSQL을 통한 데이터 쿼리 및 처리, 머신러닝 기능을 제공한다.
- 자체 또는 Mesos나 YARN을 통한 분산 처리를 지원하여 클라우드 환경에 적합하다.
- Data Source API를 통해 다양한 DB 및 파일 포맷을 지원한다.
- Zeppelin 또는 Jupyter 등의 대화형 노트북 GUI 도구를 사용할 수 있어 데이터 시각화 등이 가능하다.

7.2.4. Apache Zeppelin

o Apache Zeppelin은 노트북 형태의 반응형 인터페이스를 제공하여 웹 UI 상에서 데이터를 간편하게 처리 및 시각화하고 공유할 수 있게 해주는 오픈소스 프로젝트로, Spark 뿐만 아니라 리눅스 Shell, Hive, Cassandra, Tajo, Elasticsearch 등 다양한 데이터베이스 또는 데이터 처리 프레임워크를 지원한다.

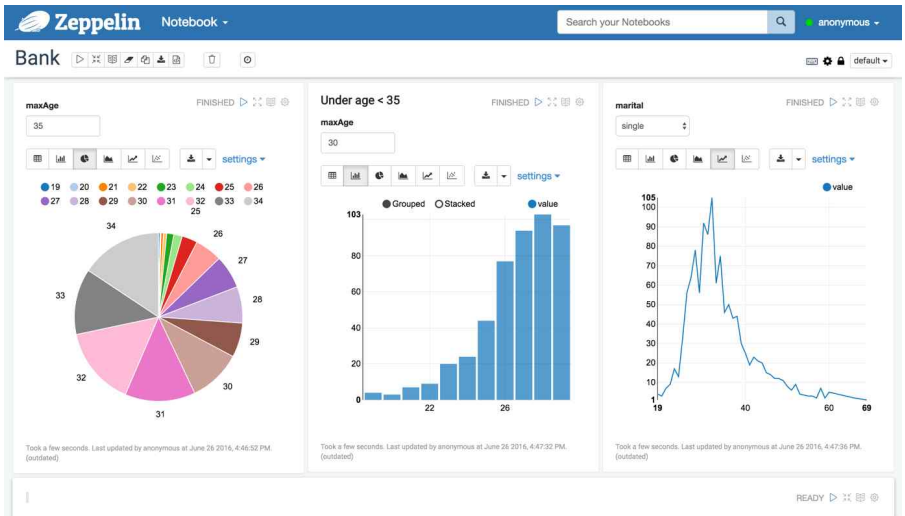


그림 61 Apache Zeppelin 화면

7.3. 설치 및 환경설정

7.3.1. 설치 개요 및 사전 준비

o Cluster Lab에서는 빅데이터 처리 플랫폼을 구성하는 소프트웨어인 Apache Mesos, Spark와 Zeppelin을 NUC 클러스터에 설치한다. 또한 좀 더 고난도에 도전하려는 실습자를 대상으로 HDFS 설치 과정을 안내한다. NUC은 같은 네트워크에 연결된 1대의 Tower와 2대의 Worker로 구성되며, Ubuntu 14.04 64bit를 운영체제로 사용한다. 각 NUC에 설치되는 소프트웨어들은 그림6과 같다.

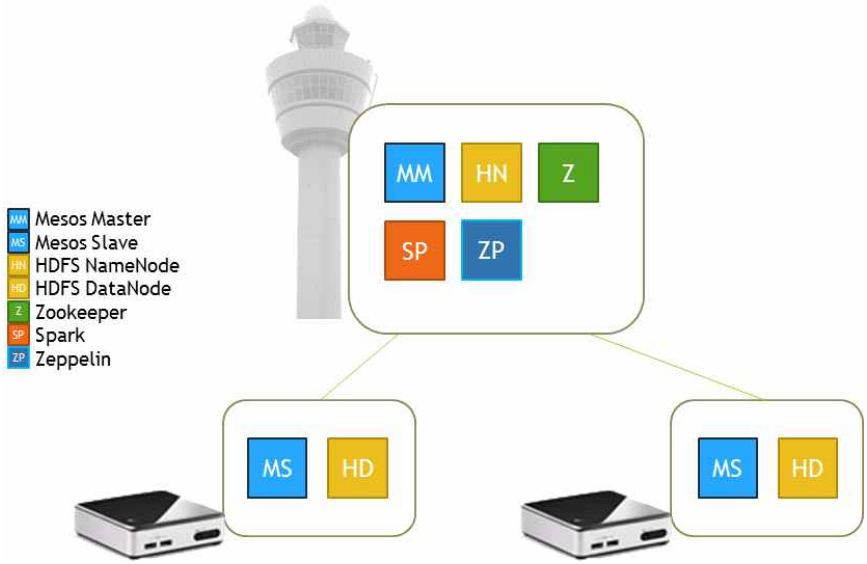


그림 62 NUC 클러스터 소프트웨어 배치 구조

o 본격적인 소프트웨어 설치를 진행하기에 앞서 이전 Lab을 참조하여 모든 NUC에 Java JDK 8을 설치해야 하고, 설치 과정을 간소화하기 위해 리눅스 사용자 이름을 hadoop으로 통일한다. 또한 /etc/hosts 파일을 수정하여 호스트 이름을 지정해주어야 한다. 이 문서에서는 호스트네임이 다음과 같이 설정되어 있다 가정하고 실습을 진행하였다.

192.168.0.1	tower
192.168.0.2	nuc01
192.168.0.3	nuc02

7.3.2. Apache Mesos 설치

o Apache Mesos 설치 과정은 tower에 Mesos Master를 설치하고 nuc01과 nuc02에 Mesos Slave를 설치하는 방식으로 진행된다. 이 문서의 설치 방법은 Mesos 0.28.1 버전을 기준으로 작성되었다.

o 우선 모든 NUC에 Mesosphere 레포지토리를 추가해준다. 이 과정을 통해 Mesos를 apt-get을 이용하여 설치할 수 있게 된다.

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
$ DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
$ CODENAME=$(lsb_release -cs)
```

```
$ echo "deb http://repos.mesosphere.io/${DISTRO} ${CODENAME} main" | sudo tee /etc/apt/sources.list.d/mesosphere.list
$ sudo apt-get -y update
```

o 다음은 tower에서 Mesos Master 설치 과정을 먼저 진행한다.

우선 apt-get을 이용한 Mesos 설치를 진행한다. 이 과정에서 클러스터 코디네이터인 Apache Zookeeper가 함께 설치된다.

```
$ sudo apt-get -y install mesos
```

마스터 노드에서는 Mesos-master 서비스만 동작하므로 Mesos-slave 서비스가 실행되지 않도록 설정해준다.

```
$ echo manual | sudo tee /etc/init/mesos-slave.override
```

Mesos Master가 사용할 IP 주소를 설정한다.

```
$ echo 192.168.0.1 | sudo tee /etc/mesos-master/ip
```

Mesos Master의 호스트 이름을 /etc/hosts 파일을 참고하여 호스트네임 또는 IP주소를 입력해준다.

```
$ echo tower | sudo tee /etc/mesos-master/hostname
```

Zookeeper 주소를 설정해준다. IP주소는 tower의 IP 주소이다.

```
$ echo zk://192.168.0.1:2181/mesos | sudo tee /etc/mesos/zk
```

Mesos 웹 UI에서 표시될 클러스터의 이름을 설정해준다.

```
$ echo SmartX-Labs | sudo tee /etc/mesos-master/cluster
```

이것으로 tower에서의 Mesos Master 설치가 완료되었며, 재부팅 후 Mesos Master가 정상적으로 작동하게 된다. 이 문서에서 소개한 설정 이외의 Mesos 옵션에 대한 추가적인 설명은 https://open.mesosphere.com/reference/mesos-master/ 에 소개되어 있다.

o 다음으로 nuc01과 nuc02에 Mesos Slave를 설치한다. 전체적인 방법은 Mesos Master를 설치할 때와 비슷하나 약간의 차이가 있다. 슬레이브 노드에서는 mesos-slave만 실행되며, mesos-master는 실행되지 않도록 설정해야 한다. 또한 본 문서에서는 Zookeeper가 마스터 노드에서만 실행되도록 하고 슬레이브 노드에서는 실행되지 않도록 하기 위하여 슬레이브 노드에서 Zookeeper를 삭제하도록 하였다.

```
$ sudo apt-get -y install mesos
$ sudo apt-get purge zookeeper
```

슬레이브 노드에서는 Mesos-slave 서비스만 동작하므로 Mesos-master가 실행되지 않도록 설정해준다.

```
$ echo manual | sudo tee /etc/init/mesos-master.override
```

각 Mesos Slave가 사용할 IP 주소를 설정한다.

```
$ echo 192.168.0.2 | sudo tee /etc/mesos-master/ip
```

앞에서 수정한 /etc/hosts 파일의 내용과 일치하도록 각각의 노드에서 사용할 호스트네임 또는 IP주소를 입력해준다.

```
$ echo nuc01 | sudo tee /etc/mesos-master/hostname
```

Zookeeper 주소를 설정해준다.

```
$ echo zk://192.168.0.1:2181/mesos | sudo tee /etc/mesos/zk
```

o 설정이 완료되면 웹 브라우저에서 http://192.168.0.1:5050 으로 접속하여 Mesos 웹 UI를 확인할 수 있다.

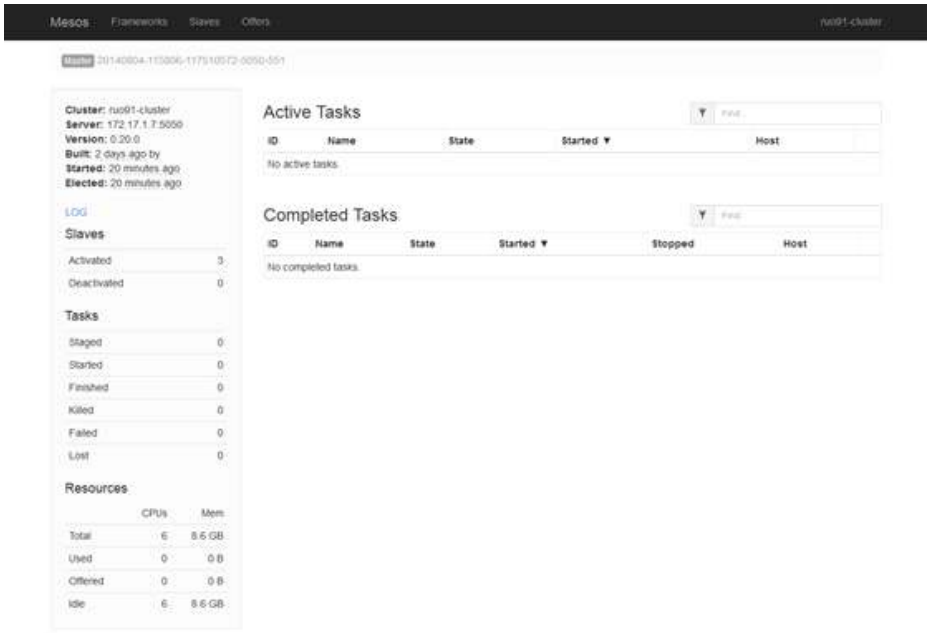


그림 63 Mesos 웹 UI 예시

7.3.3. Apache Spark 설치

o Apache Spark를 Mesos 상에서 동작하도록 설치하는 방법은 두 가지가 있다. 첫 번째는 모든 NUC에 Spark 실행 바이너리를 설치하는 방법이고, 두 번째는 Spark 실행 바이너리를 tower에만 설치하고 tgz 파일을 HDFS에 업로드하여 Spark가 실행될 때마다 Mesos Slave에서 tgz 파일을 내려받아 실행하는 방식이다. HDFS 설치 과정은 난이도가 높기 때문에 우선은 HDFS가 필요 없는 첫 번째 방법을 기술하였다. 이 문서에서는 Spark 1.6.2 버전을 기준으로 설치 및 설정 방법을 기술하였다.

o 먼저 모든 NUC에서 Spark 바이너리를 다운받아 압축을 해제한다. 이 작업은 모든 NUC이 같은 디렉토리에서 수행되어야 한다. 이 문서에서는 hadoop이라는 이름을 사용하는 사용자의 홈 디렉토리(/home/hadoop/)에서 설치를 진행한다.

```
$ wget http://mirror.apache-kr.org/spark/spark-1.6.2/spark-1.6.2-bin-hadoop2.6.tgz
$ tar xzf spark-1.6.2-bin-hadoop2.6.tgz
```

o Tower에서 Spark의 각종 설정값을 변경하기 위해 spark-env.sh 파일을 수정한다.

```
$ cd spark-1.6.2-bin-hadoop2.6/conf
$ cp spark-env.sh.template spark-env.sh
$ vi spark-env.sh
```

```
export MESOS_NATIVE_JAVA_LIBRARY=/usr/local/lib/libmesos.so
export MASTER=mesos://192.168.0.1:5050
```

o Spark 설정이 제대로 되었는지 확인하기 위해 Pyspark 쉘을 실행해 다음과 같은 예제를 실행해본다. 예제는 1부터 10000까지의 숫자 중 10 미만인 숫자를 구하는 간단한 필터링 예제이다. 그림 8과 같은 결과가 나오면 설정이 정상적으로 완료된 것이다.

```
$ cd ..
$ bin/pyspark

>>> data = range(1, 10001)
>>> distData = sc.parallelize(data)
>>> distData.filter(lambda x: x < 10).collect()
```

```
>>> distData.filter(lambda x: x < 10).collect()
16/06/29 16:57:41 INFO SparkContext: Starting job: collect at <stdin>:1
16/06/29 16:57:42 INFO DAGScheduler: Got job 1 (collect at <stdin>:1) with 2 output partitions
16/06/29 16:57:42 INFO DAGScheduler: Final stage: ResultStage 1 (collect at <stdin>:1)
16/06/29 16:57:42 INFO DAGScheduler: Parents of final stage: List()
16/06/29 16:57:42 INFO DAGScheduler: Missing parents: List()
16/06/29 16:57:42 INFO DAGScheduler: Submitting ResultStage 1 (PythonRDD[2] at collect at <stdin>:1), which has no missing parents
16/06/29 16:57:42 INFO MemoryStore: Block broadcast_1 stored as values in memory (estimated size 3.4 KB, free 9.1 KB)
16/06/29 16:57:42 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (estimated size 2.3 KB, free 11.4 KB)
16/06/29 16:57:42 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on 192.168.88.147:37555 (size: 2.3 KB, free: 511.1 MB)
16/06/29 16:57:42 INFO SparkContext: Created broadcast 1 from broadcast at DAGScheduler.scala:1006
16/06/29 16:57:42 INFO DAGScheduler: Submitting 2 missing tasks from ResultStage 1 (PythonRDD[2] at collect at <stdin>:1)
16/06/29 16:57:42 INFO TaskSchedulerImpl: Adding task set 1.0 with 2 tasks
16/06/29 16:57:42 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 2, nuc08, partition 0,PROCESS_LOCAL, 17269 bytes)
16/06/29 16:57:42 INFO TaskSetManager: Starting task 1.0 in stage 1.0 (TID 3, nuc07, partition 1,PROCESS_LOCAL, 16802 bytes)
16/06/29 16:57:42 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on nuc08:40305 (size: 2.3 KB, free: 511.1 MB)
16/06/29 16:57:42 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 2) in 40 ms on nuc08 (1/2)
16/06/29 16:57:42 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on nuc07:33340 (size: 2.3 KB, free: 511.1 MB)
16/06/29 16:57:42 INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 3) in 446 ms on nuc07 (2/2)
16/06/29 16:57:42 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed, from pool
16/06/29 16:57:42 INFO DAGScheduler: ResultStage 1 (collect at <stdin>:1) finished in 0.447 s
16/06/29 16:57:42 INFO DAGScheduler: Job 1 finished: collect at <stdin>:1, took 0.464302 s
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

그림 64 Pyspark 쉘에서 예제 실행 결과

7.3.4. Apache Zeppelin 설치

o Zeppelin의 설치도 Spark와 비슷하게 진행된다. 이 문서에서는 앞과 마찬가지로 홈 디렉토리에서 작업하는 것으로 하였으며, Zeppelin 0.5.6을 기준으로 작성되었다.

o Zeppelin의 경우 tower에서만 설치를 진행한다. tower의 홈 디렉토리에서 다음과 같이 바이너리를 다운받아 압축을 해제한다.

```
$ wget http://mirror.apache-kr.org/incubator/zeppelin/0.5.6-incubating/zeppelin-
```

0.5.6-incubating-bin-all.tgz

```
$ tar xzf zeppelin-0.5.6-incubating-bin-all.tgz
```

o Zeppelin은 다운받은 바이너리 자체로도 Spark를 내부에 포함하고 있으나, 앞서 Mesos를 스케줄러로 사용하도록 설정한 Spark를 사용하기 위해 다음과 같이 zeppelin-env.sh 파일을 수정한다.

```
$ cd zeppelin-0.5.6-incubating-bin-all/conf
$ cp zeppelin-env.sh.template zeppelin-env.sh
$ vi zeppelin-env.sh

export SPARK_HOME=/home/hadoop/spark-1.6.2-bin-hadoop2.6
```

o Zeppelin 데몬을 실행하고 웹 브라우저로 http://192.168.0.1:8080에 접속하여 Zeppelin 웹 UI가 정상적으로 실행되는지 확인한다.

```
$ cd ~/zeppelin-0.5.6-incubating-bin-all
$ bin/zeppelin-daemon.sh start
```

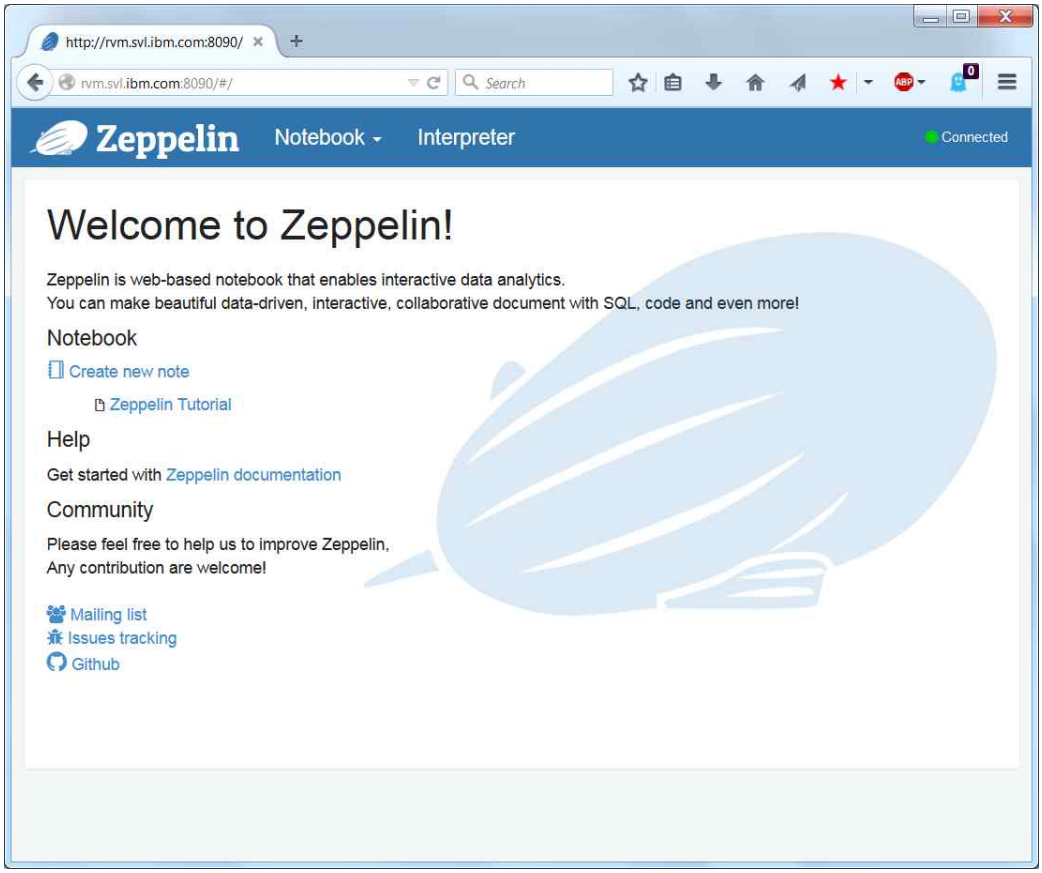


그림 65 Zeppelin 웹 UI 화면

o Zeppelin 웹 UI에서 Zeppelin Tutorial로 들어가서 노트북을 실행해 정상적으로 작동하는지 확인한다.

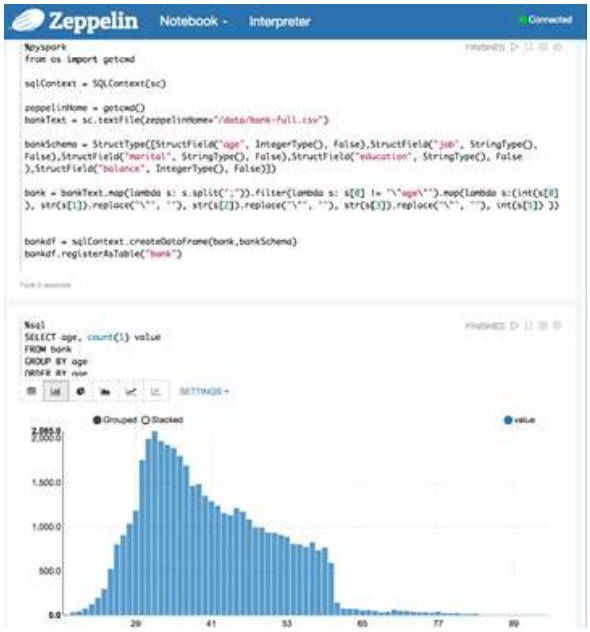


그림 66 Zeppelin Tutorial이 정상적으로 실행된 모습

7.3.5. Apache Hadoop HDFS 설치

o HDFS의 설치 과정은 앞서 설명한 내용들에 비해 복잡하기 때문에 앞의 과정을 문제 없이 빠른 시간 내에 완료한 사람을 대상으로 진행된다.

o HDFS는 노드 간에 SSH를 통해 통신하므로 이를 먼저 설정해주어야 한다.

모든 NUC에서 다음과 같이 키를 생성한다.

```
$ ssh-keygen -t rsa
$ cp /home/hadoop/.ssh/id_rsa.pub /home/hadoop/.ssh/authorized_keys
```

tower에서 키 권한을 설정하고 인증키 파일을 nuc01과 nuc02로 복사한다.

```
$ cd .ssh
$ chmod 644 authorized_keys
$ scp authorized_keys hadoop@nuc01:~/.ssh/
$ scp authorized_keys hadoop@nuc02:~/.ssh/
```

tower에서 nuc01과 nuc02에 암호를 입력하고 ssh로 접속할 수 있는지 확인해본다.

o 다음으로, 모든 NUC에서 Hadoop을 다운받아 압축을 해제하고 설정을 변경해준

다.

```
$ wget http://mirror.apache-kr.org/hadoop/common/hadoop-2.7.2/hadoop-2.7.2.tar.gz
$ tar -xvzf hadoop-2.7.2.tar.gz
$ sudo mv hadoop-2.7.2 /usr/local/hadoop
```

tower에서 설정 파일이 위치한 디렉토리로 이동한다.

```
$ cd /usr/local/hadoop/etc/hadoop
```

다음 파일들의 내용을 수정해야 한다.

- hadoop-env.sh 파일

```
export JAVA_HOME=/usr/lib/jvm/java-8-oracle
```

- core-site.xml 파일

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://tower:9000/</value>
    </property>
</configuration>
```

- hdfs-site.xml 파일

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>2</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:///usr/local/hadoop/namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:///usr/local/hadoop/datanode</value>
    </property>
</configuration>
```

- slaves 파일
 nuc1
 nuc2

tower에서 수정한 설정 파일들을 nuc01과 nuc02로 복사한다.
 $ cd ..
 $ scp –r hadoop hadoop@nuc01:/usr/local/hadoop/etc/
 $ scp –r hadoop hadoop@nuc02:/usr/local/hadoop/etc/

nuc01과 nuc02에서 DataNode 디렉토리를 생성한다.
 $ mkdir /usr/local/hadoop/datanode

모든 NUC에서 /etc/environment 파일을 수정하여 환경변수를 설정한다. 파일의 끝에 따옴표를 지우고 다음을 추가해준다.
:/usr/local/hadoop/bin“

재부팅한다.

o 이제 HDFS가 정상적으로 실행되는지 확인한다. HDFS의 실행 방법은 다음과 같다.

 $ hdfs namenode –format
 $ hadoop/sbin/start-dfs.sh

HDFS에서 디렉토리를 생성하고 파일을 업로드하여 정상적으로 동작하는지 확인한다.

 $ hadoop fs –mkdir /user
 $ hadoop fs –put ~/hadoop-2.7.2.tar.gz /user/
 $ hadoop fs –ls hdfs://tower:9000/user/

o HDFS가 정상적으로 설치되었다면 Spark를 HDFS를 이용해 동작하도록 설정해준다.

 $ hadoop fs –put spark-1.6.2-bin-hadoop2.6.tgz /user/

spark-env.sh에 다음 내용을 추가해준다.
export SPARK_EXECUTOR_URI=hdfs://<TOWER_IP>:9000/user/spark-1.6.2-bin-hadoop2.6.tgz

마찬가지로 Pyspark 쉘에서 예제를 실행해 정상적으로 작동하는지 확인한다.

7.4. 결론

o 본 기술문서는 빅데이터 처리를 위해 실제로 어떤 소프트웨어들이 어떻게 구성되어 사용되는지에 대하여 관련 내용을 처음 접하는 사람들을 대상으로 작성되었으며, 리눅스 환경에서 Apache Mesos, Apache Spark 및 Zeppelin을 직접 설치해 보는 실습이다.

8. Analytics Lab

8.1. Analytics Lab Background

8.1.1. 목적 및 개요

o Analytics Lab은 앞서 Cluster Lab에서 구성한 환경을 이용해 데이터 분산 처리 원리에 대해 이해하고 Zeppelin 노트북을 이용해 직접 간단한 데이터 처리 예제를 실행해보는 것을 목표로 한다. 이를 통해 클라우드에서 빅데이터 처리가 어떻게 이루어지는지와 실제 데이터 과학자들 또는 프로그래머들의 데이터 작업 환경을 간단히 체험해볼 수 있다.

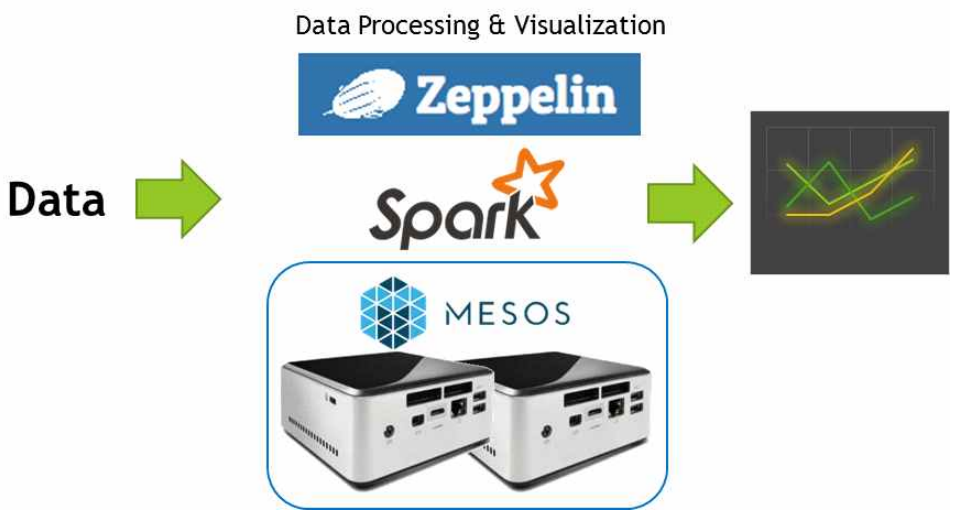


그림 67 Analytics Lab Overview

8.1.2. Map & Reduce

o Map & Reduce는 분산 자원 환경에서 데이터를 분산 처리하는 방법론이다. Map & Reduce는 데이터를 적절히 분산하고, 분산된 데이터 각각을 병렬로 처리한 후, 계산된 결과 데이터를 합치는 과정으로 이루어져 있다. 그림 2의 word count 예제를 보면, 단어들을 나누고, map 함수로 각 단어의 수를 1로 초기화한 다음, 같은 단어끼리 묶고 reduce 함수를 이용해 같은 단어끼리 수를 합쳐 최종적으로 단어의 수가 각각 몇 개인지 알 수 있다.

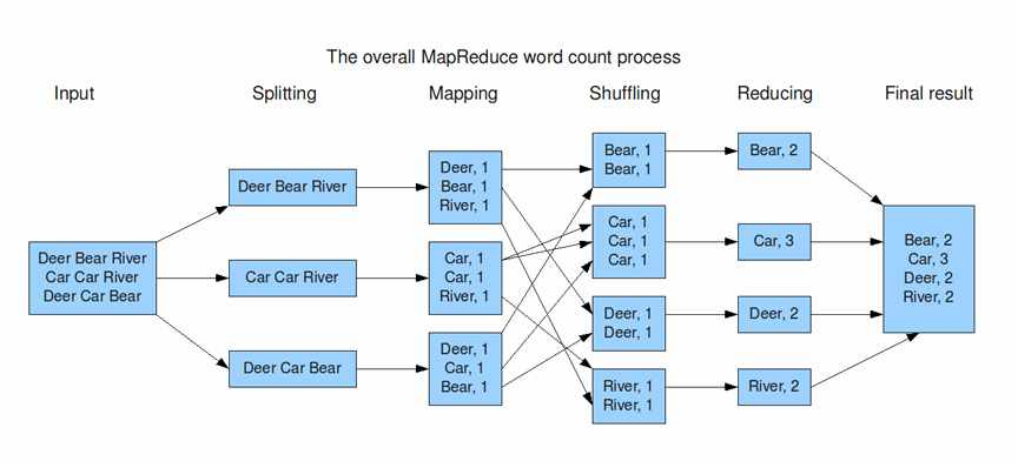


그림 68 MapReduce를 사용한 word count 예시

8.1.3. Apache Spark와 Apache Zeppelin

o 이 랩에서는 빅 데이터 처리 프레임워크로 Apache Spark를 사용하며, 이것을 위한 사용자 인터페이스로 Apache Zeppelin을 사용한다. Spark는 기존의 Hadoop이나 Apache Storm 같은 프레임워크에 비해 프로그래밍이 간단하기 때문에 이 실습의 대상이 되는 초보자들이 쉽게 예제 코드를 이해할 수 있다는 장점이 있다. 또한 Zeppelin은 복잡하게 느껴질 수 있는 IDE보다 좀 더 사용자 친화적이고 가벼우며 데이터 분석을 위한 기능들을 손쉽게 쓸 수 있도록 제공하므로 실습용으로 적합하다.

o 앞서 Cluster Lab에서 Spark와 Zeppelin에 대해 설명하고 설치 과정에 대해 설명하였다. 그렇게 설정된 환경에서 Zeppelin을 실행하면 Zeppelin 프레임워크가 Mesos 상에서 실행된다. Zeppelin 데몬 실행 직후에는 Zeppelin은 자원을 할당받지 않은 상태로 있으나 Zeppelin에서 Spark 작업을 실행하여 Spark가 실제로 RDD 연산을 수행하게 되면 그제서야 Mesos에서 자원을 할당받게 된다.

8.2. Spark와 Zeppelin을 이용한 실습

8.2.1. 실습 1: Pyspark on Zeppelin

o Spark는 Scala, Java, Python, R 언어를 지원한다. 그 중 프로그래밍 경험이 많지 않은 실습자들에게는 Python이 사용하기 비교적 쉬운 언어이다. 따라서 본 문서에서는 Python 언어로 실습을 진행하며, Spark의 Python 지원 모듈인 Pyspark를 이용한다. 본 실습 1은 Scala로 작성되어 있는 Zeppelin Tutorial 코드를 Pyspark로 변환한 노트북을 이용해 진행된다.

o 먼저 tower에서 Zeppelin 데몬을 실행한다.

```
$ bin/zeppelin-daemon.sh start
```

Zeppelin 데몬이 실행중인 상태에서 웹 브라우저로 http://192.168.0.1:8080에 접속하여 Zeppelin 웹 UI를 연다.

Zeppelin에서 사용하는 Spark는 Mesos 프레임워크로 동작하도록 구성하였기 때문에 Mesos 웹 UI에서 Framework 메뉴로 들어가면 Zeppelin이 프레임워크로 등록되어 있는 것을 확인할 수 있다.

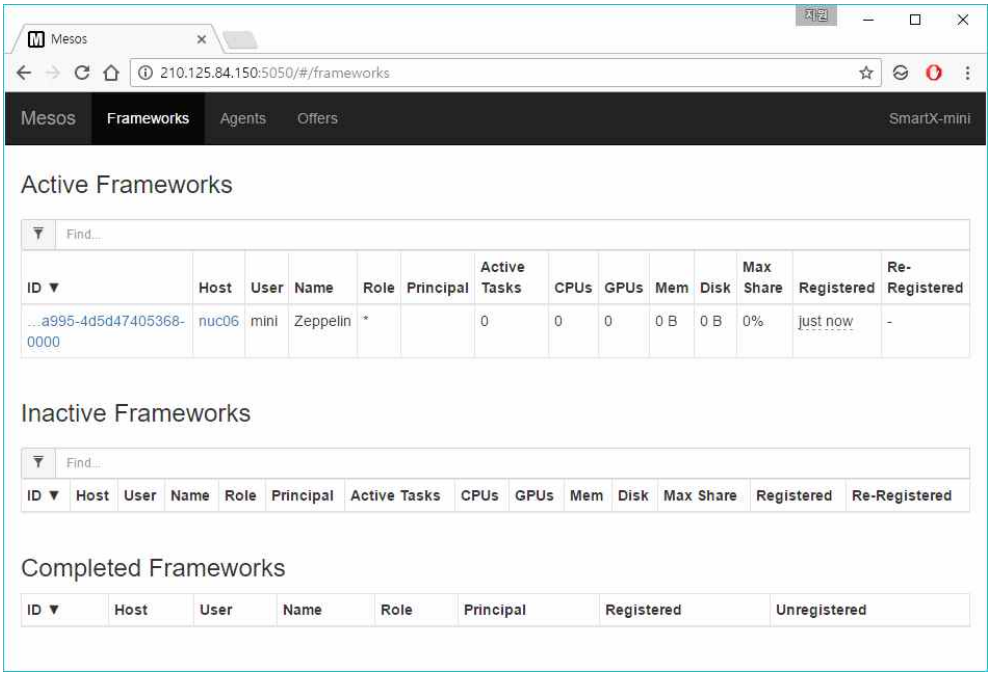


그림 69 Mesos에서 Zeppelin 프레임워크 확인

o 새 노트를 추가하려면 상단의 Notebook -> + Create new note를 선택하면 된다. 그러나 본 실습에서는 실습 시간을 고려하여 미리 작성된 Pyspark 튜토리얼 노트북을 Github에 올려놓았다. Analytics_Lab-Pyspark.json 파일을 다운받고 zeppelin에서 Import note > Choose a JSON here을 순서대로 눌러 노트북 JSON 파일을 업로드하면 노트북이 만들어진다.

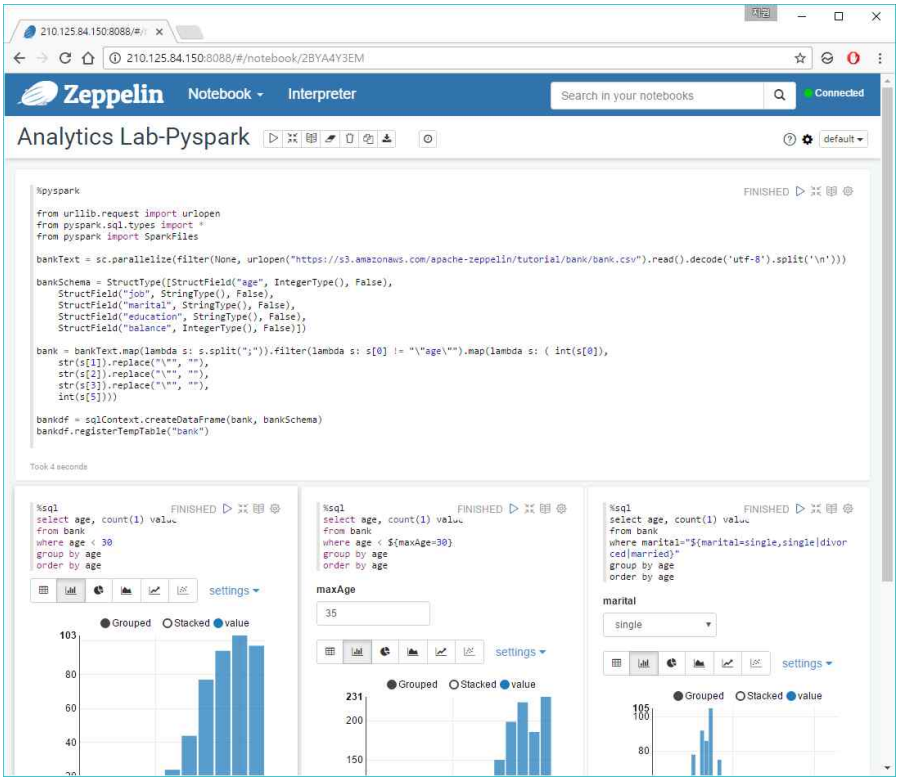


그림 70 Analytics Lab-Pyspark 노트북을 업로드하여 실행한 모습

이제 노트북 제목 옆의 Run all paragraph 버튼을 누르면 Scala로 작성되어 있던 Zeppelin Tutorial 코드와 동일한 결과를 출력한다.

8.2.2. 실습 2: Wordcount

o 본 실습에서는 앞에서 빅데이터 처리 원리를 설명할 때 예로 든 word count 예제를 Pyspark 코드로 작성하여 실행해본다. 이 실습 또한 프로그래밍이 익숙하지 않은 사람들을 위하여 미리 작성되어 Github에 업로드 되어있는 노트북 파일을 사용한다.

o 본 실습에서 사용된 소스코드를 보면 먼저 데이터 텍스트 파일을 불러와서 모든 특수문자를 제외하고 대문자를 소문자로 변경한다. 그리고 띄어쓰기를 기준으로 각 단어를 구분한 후, map 함수를 이용해 각각의 단어들을 (단어, 1)의 형태로 만든다. 이러한 key-value 데이터들을 reduce 함수를 이용해 같은 단어(같은 키)들의 값을 더하여 (단어, 단어 수)의 결과값을 얻는다.

o Github에서 Analytics_Lab-Wordcount.json 파일을 다운받고 Zeppelin에서 이전 실습과 같은 방법으로 불러온다. 불러온 노트북을 열고 Run all paragraph를 누르면 노트북이 순차적으로 실행이 되고 단어 개수가 표로 나타난다.

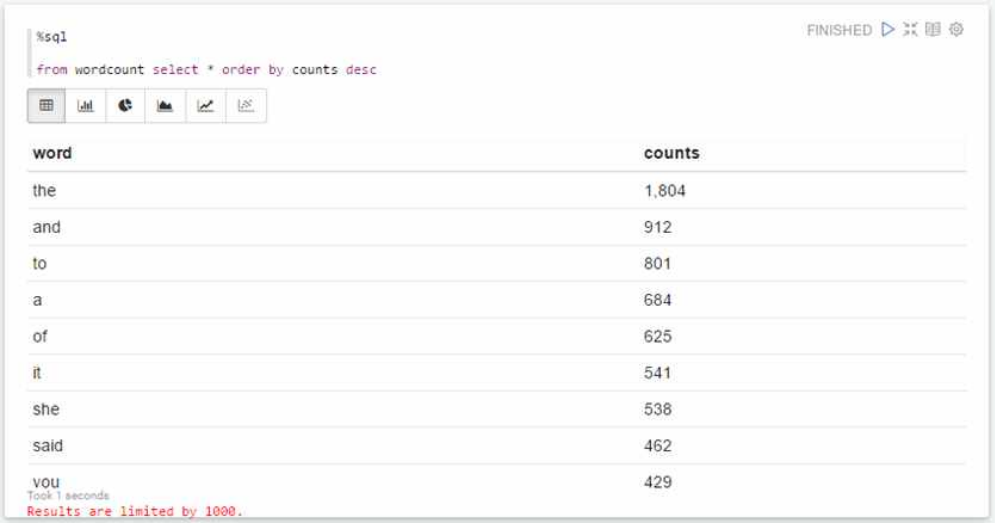


그림 71 Analytics Lab-Wordcount 실행 결과

표의 좌측 상단에는 표를 다양한 종류의 그래프로 시각화할 수 있는 도구가 제공된다. 그림 6은 막대그래프를 선택했을 때의 모습이다.

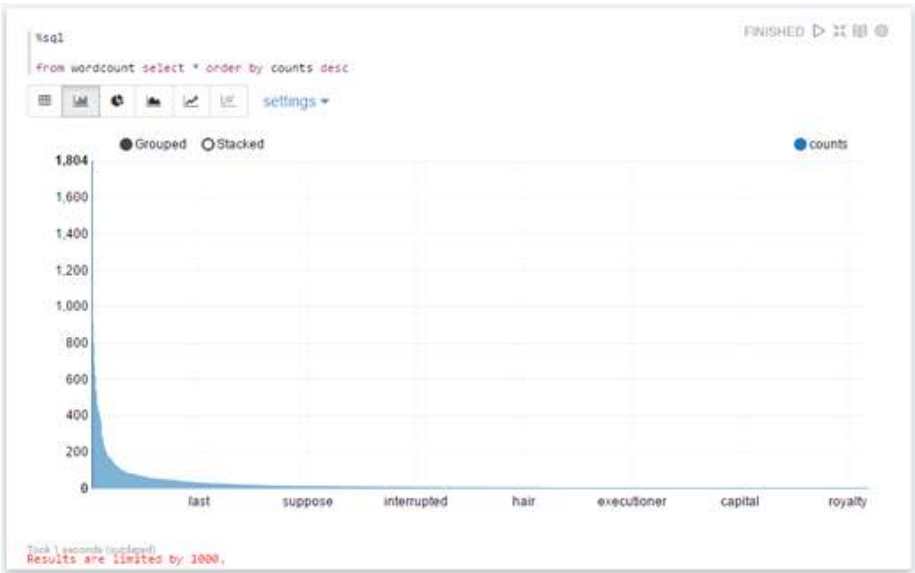


그림 72 Analytics Lab-Wordcount 실행 결과 출력 형태를
막대그래프로 변경

8.3. 결론

o 본 기술문서 Analytics Lab에서는 분산 처리 환경과 Zeppelin 노트북을 이용해 리눅스 프로그래밍 경험이 많지 않은 사람들도 직접 간단한 빅데이터 처리 예제를 실습할 수 있도록 구성하였다.

9. SDN Lab

9.1. SDN Lab Background

9.1.1. 목적 및 개요

o SDN Lab은 SDN Network에 대한 이해와 지금껏 구성된 SmartX-mini Playground에서 SDN 제어기를 통해 SmartX-mini Playground의 Network를 전반적으로 다뤄보는 과정을 담고 있다. 기존의 네트워크 장비인 스위치의 대한 이해가 필요하며, SDN Controller를 통해 네트워크 플로우를 직접 다뤄보고, OpenFlow를 이해하는 것에 초점을 둔다.

o 본 기술문서의 SDN Part의 구성환경은 그림 1과 같다. ONOS(Open Network Operating System) SDN Controller을 사용하며, 가상 네트워크 환경인 Mininet을 구축하여 하나의 물리적인 머신(Intel NUC)에서 Lab을 진행한다. Mininet 환경에서 ONOS의 Intent Application을 사용해 플로우의 추가/제거를 다뤄본다.

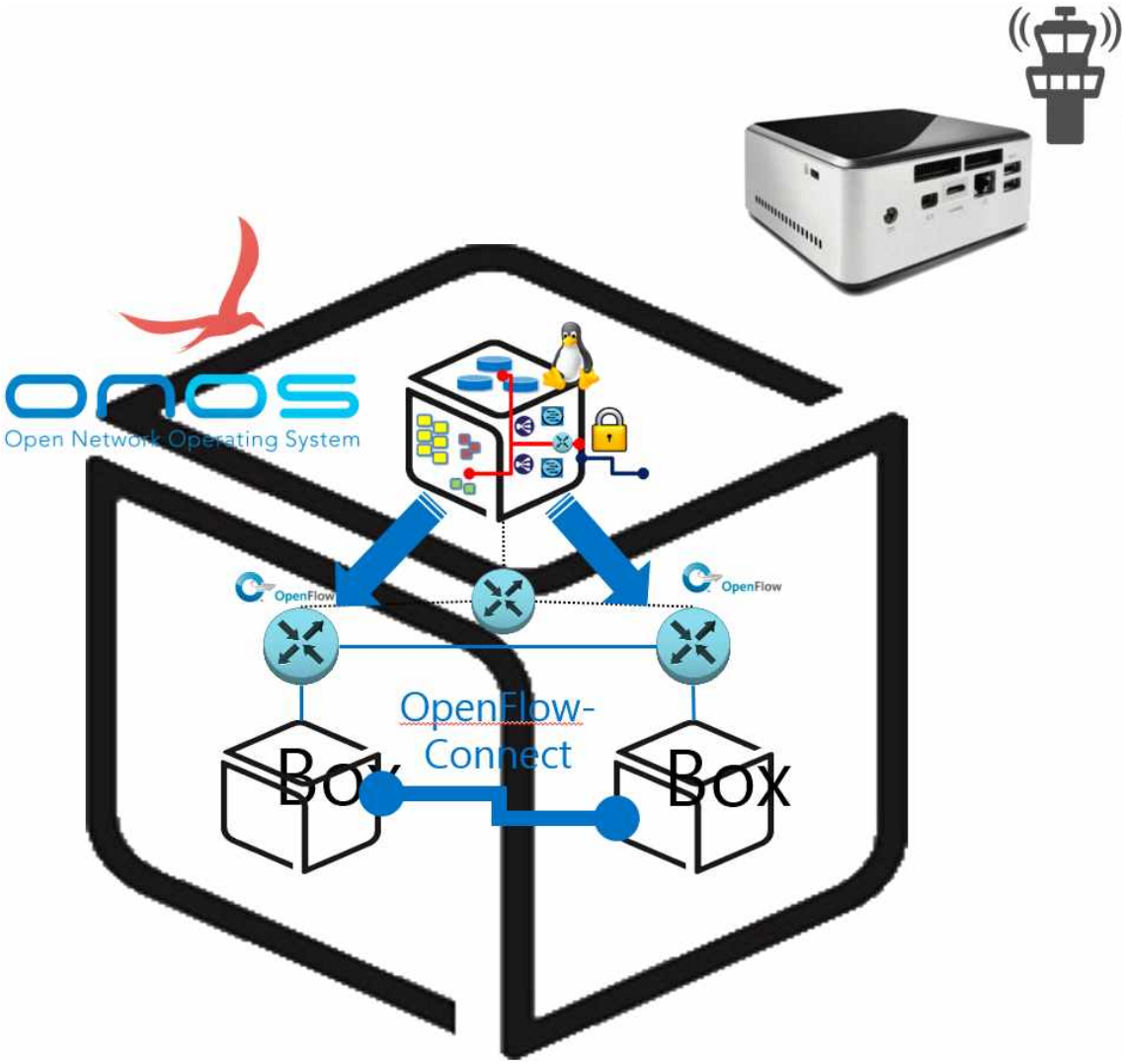


그림 73 SDN Lab overall outline.

9.1.2. SDN(Software-Defined Network)

o SDN(Software-Defined Networking, SDN)은 개방형 API(OpenFlow)를 통해 네트워크의 트래픽 전달 동작을 소프트웨어 기반 제어기에서 제어/관리하는 접근 방식이다. 트래픽 경로를 지정하는 Control Plane과 트래픽 전송을 수행하는 Data Plane이 분리되어 있다. 따라서 네트워크의 세부 구성정보에 얽매이지 않고 요구사항에 따라 네트워크를 관리 할 수 있다.

o 중앙의 SDN 제어기가 Control Plane을 제어하면서 기존 네트워크 장비는 Data Plane 기능만을 담당하게 된다. Control Plane이 분리됨으로써, SDN이 기존 네트워크보다 가지는 장점이 있다. 여러 스위치들을 중앙에서 한 번에 관리할 수 있어 네트워크 상황에 따른 제어가 손쉽게 이뤄질 수 있다. 또한 관리자가 네트워크를 손쉽게 프로그래밍 할 수 있다. Control Plane은 SDN의 두 가지 요소와 상호작용하는데 하나는 Application이고, 다른 하나는 하드웨어에 구현된 추상화 계층이다. 그림 2와 같이 Control Plane 기능을 통해서 Application은 네트워크의 다양한 정보를 얻을 수 있고, 반대로 네트워크 역시 Application 요구 사항 등의 정보를 얻을 수 있다. 이러한 핵심적인 역할 때문에, Control Plane은 종종 네트워크 운영체제(Network OS)라 호칭되고 있다.

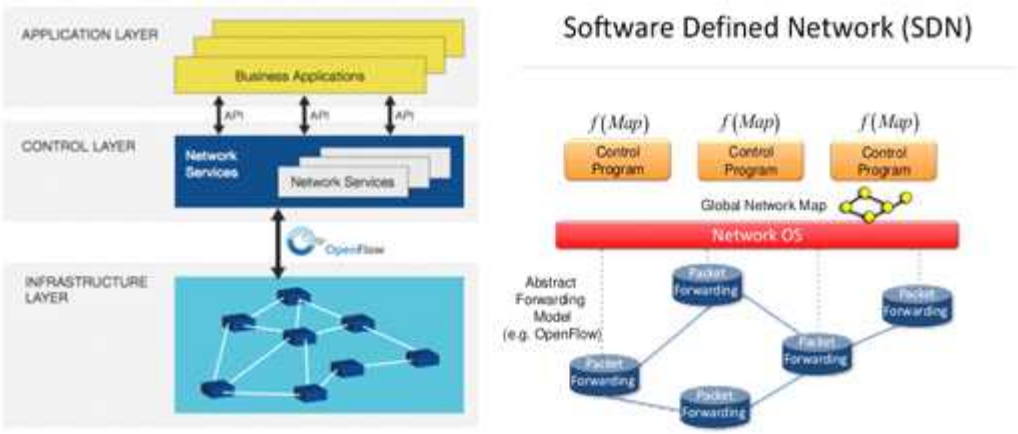


그림 74 SDN Architecture (https://www.opennetworking.org/sdn-resources/sdn-definition)

9.1.3. OpenFlow

o OpenFlow는 SDN 환경에서 Control Plane과 Data Plane 간의 통신 프로토콜로 제어기와 하드웨어 장비 간의 통신 내용을 정의하는 SDN의 기반이 되는 기술이다. 네트워크 스위치나 라우터의 Forwarding 계층과 제어 계층 기능을 분리하고 이들 간의 통신을 위한 프로토콜을 제공한다. OpenFlow는 데이터 경로를 설정함으로써 네트워크 장비의 프로그램 가능성을 제공하여 트래픽을 어떤 네트워크 장비 속으로 흐르게 할 것인지 정의가 가능하다. OpenFlow는 개방형 네트워크 재

단(ONF: Open Networking Foundation)에서 관리되며, ONF는 비영리, 상호 이익을 바탕으로 하는 국제기구로 OpenFlow는 ONF라는 단체를 통하여 업계의 주요 업체들을 후원세력으로 갖는다.

9.1.4. ONOS(Open Network Operating System)

o ONOS(Open Network Operating System)는 분산형 SDN 제어기 중 가장 대표적인 제어기다. 기존의 중앙집중식 SDN 제어기 구조의 문제점을 극복하고, 실제 대규모 네트워크 환경에 적합한 구조로 설계되어 있다. 그림 3은 ONOS의 구조를 개념적으로 나타내는 개념도이다. 그림 3에서 볼 수 있듯이, 하단에 위치한 네트워크 스위칭 장비들을 분산형 구조를 갖는 ONOS 제어기가 제어하는 구조이다. ONOS의 계층은 크게 어플리케이션 계층(Apps), 분산 코어 계층(Distributed Core), 프로바이더 계층(Providers), 프로토콜 계층(Protocols)으로 구성 된다. 프로토콜 계층은 SDN 기능이 있는 다양한 네트워크 장치들과의 통신을 위한 계층으로 대표적인 OpenFlow가 있다. 프로바이더 계층은 각 ONOS 제어기 마다 독립적으로 동작하는 계층으로 네트워크 장치들에게 명령을 지시한다. 분산 코어 계층은 SDN 네트워크를 제어 및 관리할 수 있는 다양한 기능들이 정의된 계층이다. 마지막으로 어플리케이션 계층은 그래픽 유저 인터페이스(GUI: Graphic User Interface)를 통해, 분산 코어 계층에 정의된 네트워크 제어 및 관리 기능들을 수행 할 수 있도록 다양한 어플리케이션이 정의된 계층이다.

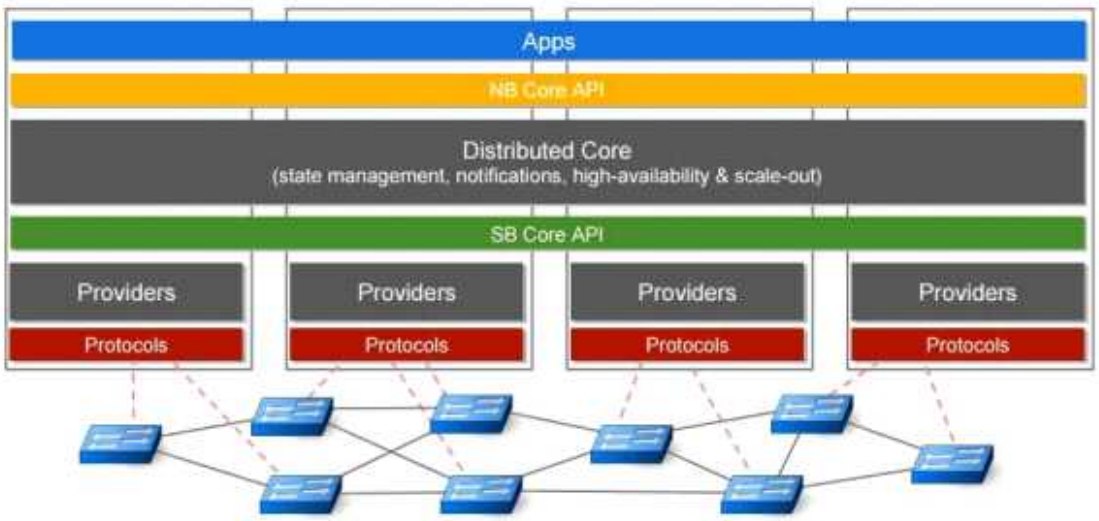


그림 75 ONOS 제어기 구조

9.2. Setting & Configuration

9.2.1. ONOS SDN 제어기

o ONOS(Open Network Operating System) 제어기를 설치하는 방법은 두 가지가 있다.

1. Apache Maven, Apache Karaf를 이용해 직접 Build하는 방법
2. ONOS에서 제공하는 버전 별 Image 파일을 다운받는 법

본 기술 문서에서는 ONOS 설치에 초점을 두기보다 사용에 초점을 두고 있기 때문에 2번 방식을 이용하여 설치를 진행한다.

ONOS(Open Network Operating System)을 설치하기 위해서는 가장 처음 Java JDK를 설치해야 한다. 특히 ONOS를 원활하게 지원하기 위해서는 Java JDK 1.8 버전 이상을 설치해야 정상적으로 설치가 가능하다. 모든 설치는 Intel NUC 머신에서 진행 된다.

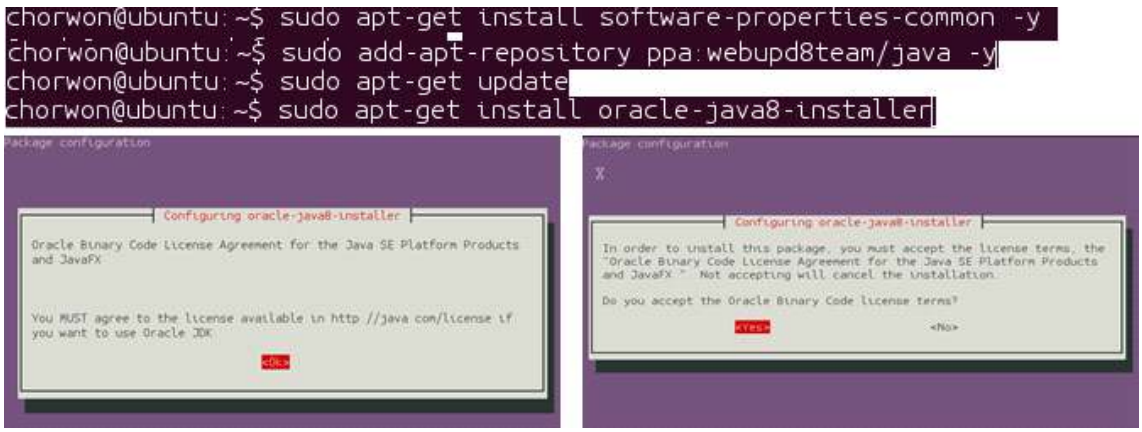


그림 76 Java JDK Install

o Java JDK 1.8버전을 그림 4의 명령어를 이용해서 설치 할 수 있다. 설치를 완료한 후, 그림 5와 같이 Java 환경변수 설정을 다음과 같이 진행한다. 이후, Java가 원활히 설치했는지 확인하기 위해서는 "$ Java -version" 명령어를 통해 설치된 Java의 버전을 확인 할 수 있다.

그림 77 Java 환경 변수 설정 및 버전 확인

그림 78 ONOS Image install & execute

o 본 기술 문서에서는 ONOS Wiki에서 제공하는 버전 별 이미지 파일 중에서 1.5.0, 코드네임 Falcon 버전을 받아 설치를 진행한다. 그림 6과 같이 압축파일을 받아 압축을 푼 후, 해당 디렉토리로 이동해 "$ bin/onos" 명령어를 통해 ONOS 제어기를 실행시킨다.

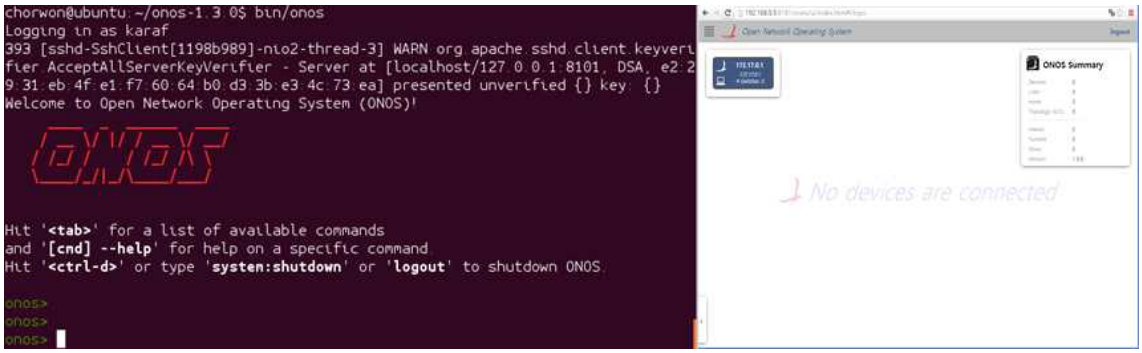


그림 79 ONOS 제어기 구동

o ONOS 제어기가 성공적으로 구동이 되면 그림7의 경우처럼 CLI 상에서 ONOS 마크가 나타나면서 ONOS CLI로 접근이 가능해진다. 또한 웹 브라우저(크롬 추천)를 통해서도 확인이 가능하며, 웹 브라우저의 접속 주소는 "http://[Controller IP]:8181/onos/ui/login.html#/"이다. [ControllerIP]는 ONOS 제어기의 IP이며, Default ID/Password는 "karaf/karaf"이다. 웹 브라우저 역시 정상적으로 설치가 되었으면 그림 7과 같이 나타난다.

9.2.2. Mininet Install & Setting

o 본 기술문서는 SDN을 직접 다뤄보는 것에 초점을 두고 있으며, 이를 가장 잘 실행하기 위한 측면으로 Mininet을 이용 할 수 있다. Mininet은 네트워크 Emulator로써 가상의 호스트, 스위치, 링크 등을 생성해준다. 또한 Openvswitch 기반으로 만들어졌기 때문에 OpenFlow 프로토콜도 지원이 가능해, SDN 실험을 할 때 가장 널리 쓰이는 도구 중 하나이다. 기본적으로 CLI를 제공하고, Python API를 통해 사용자가 원하는 복잡한 토폴로지도 생성이 가능하다.

```
mini@nuc12:~/onos-1.5.0$ sudo apt-get update && sudo apt-get upgrade && sudo apt-get install git
 sudo apt-get install mininet
git clone git://github.com/mininet/mininet
cd mininet/util/
./install.sh -a
```

그림 80 Mininet install & execute

o 그림 8과 같이 git을 통해 Mininet을 clone 받아 손쉽게 설치가 가능하다. 설치가 완료된 후에는 mininet/util/ 디렉토리로 이동해 mininet을 실행시킬 수 있다.

```
mini@nuc12:~/onos-1.5.0/mininet/util$ sudo mn --topo tree,2,3 --controller=remote,ip=127.0.0.1,port=6633
[sudo] password for mini:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s2, h1) (s2, h2) (s2, h3) (s3, h4) (s3, h5) (s3, h6) (s4, h7) (s4, h8) (s4, h9)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet>
```

그림 81 mininet 명령어를 통해 mininet topology 형성

o mininet 설치가 완료되었으면, 그림 9의 “mn” 명령어를 이용해 mininet topology를 생성 할 수 있다. “--topo tree,2,3” 은 topology를 어떻게 구성할 것인가를 정해주는 옵션이고, “controller” 옵션은 설치한 ONOS 제어기를 가리키는 옵션이다. 그림 9와 같이 명령어를 알맞게 입력하게 되면, mininet topology가 생성되며, 9개의 host와 4개의 switch를 구성하게 된다.

9.2.3. Mininet을 이용한 ONOS SDN 제어기 사용 예제: Reactive Forwarding App

o Mininet을 통해 가상 호스트와 스위치를 생성하더라도, ONOS 웹브라우저에서 Mininet의 가상 호스트와 스위치를 식별할 수 없다. 기본적으로 ONOS는 70여 개의 North-bound-Interface 어플리케이션들이 존재하고, 각각의 용도에 맞게 어플리케이션들을 실행시켜야 원하고자 하는 것을 할 수 있다. 본 기술문서에서 다루는 예제는 ONOS의 Intent 어플리케이션을 사용해보는 것과 Reactive Forwarding 어플리케이션과의 차이점을 알아보는 예제이다.

o 첫 번째 단계로 Mininet의 가상 호스트와 가상 스위치를 ONOS 제어기가 식별하기 위해서는 그림 10과 같이 몇 가지 어플리케이션들을 활성화 시켜야 한다.

Default Device Drivers 어플리케이션은 가상 스위치를 식별하기 위한 기본 어플리케이션, Host Location Provider, Host Mobility App 어플리케이션은 가상 스위치에 연결된 호스트들을 식별하기 위함이다. 또한 LLDP Link Provider은 각 스위치의 연결을 위한 어플리케이션으로 스위치는 이 어플리케이션을 통해 토폴로지를 형성 할 수 있다. OpenFlow Provider 어플리케이션은 ONOS 제어기가 SBI 인터페이스로 OpenFlow를 사용하기 위해 쓰인다. 따라서 그림 10과 같이 6개의 어플리케이션들을 활성화시키게 되면 웹브라우저에서 그림 11과 같이 토폴로지가 생성된다.

Applications (70 total)

	Icon	Title	App ID	Version	Category	Origin
✓		Default Device Drivers	org.onosproject.drivers	1.5.0	Drivers	ON.Lab
✓		Host Location Provider	org.onosproject.hostprovider	1.5.0	Provider	ON.Lab
✓		Host Mobility App	org.onosproject.mobility	1.5.0	Utility	ON.Lab
✓		LLDP Link Provider	org.onosproject.lldpprovider	1.5.0	Provider	ON.Lab
✓		OpenFlow Provider	org.onosproject.openflow-base	1.5.0	Provider	ON.Lab
✓		Proxy ARP/NDP App	org.onosproject.proxyarp	1.5.0	Traffic Steering	ON.Lab
		ACL App	org.onosproject.acl	1.5.0	Security	DLUT
		Authentication App	org.onosproject.aaa	1.5.0	Security	ATT

그림 82 ONOS 제어기의 NBI 어플리케이션

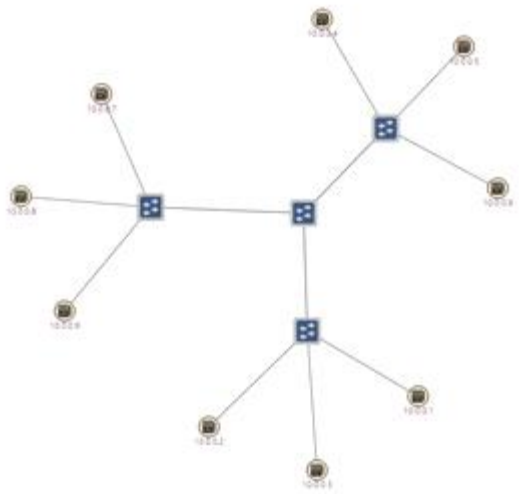


그림 83 ONOS GUI Topology

o ONOS 웹 브라우저를 통해 가상 호스트와 스위치가 서로 연결된 Topology를 확인 할 수 있다. 두 번째 단계로 Mininet CLI를 통해서 각 호스트끼리 서로 ICMP

Packet(Ping)을 보내 통신이 가능한지를 확인한다. 그림 11과 같이 Mininet CLI에서 “pingall” 이라는 명령어를 실행한다. “pingall” 명령어는 형성된 가상 네트워크의 모든 호스트들이 서로 ICMP packet을 보내 통신이 가능한지를 확인해보는 명령어다.

그림 84 Mininet “pingall” 명령어 결과

o 현재까지 본 기술문서의 순서대로 실행을 했다면, “pingall” 명령어를 입력했을 때, 모든 ICMP packet이 제대로 전송되지 않는 것을 알 수 있다. 통신이 이뤄지지 않는 이유는 가상 스위치들에게 각 packet을 어떻게 처리해야하는지에 대한 Flow가 없기 때문이다. SDN에서 기본적으로 스위치가 packet을 받았을 때, 스위치가 가지고 있는 Flow Table과 비교해 해당되지 않으면 packet을 SDN 제어기로 보내게 된다. packet을 받은 SDN 제어기는 그에 맞는 Flow를 스위치들에게 내려 해당 packet이 전송되도록 제어한다. 예제에서는 ONOS 제어기가 ICMP packet을 어떻게 처리할지 모르기 때문에 스위치들에게 Flow를 내릴 수 없기 때문에 전송이 이루어지지 않는다.

o 원활한 전송을 지원하기 위해서는 Flow를 내릴 필요가 있다. 우선, ONOS 제어기에서 제공하는 70여개의 어플리케이션들 중 Reactive Forwarding App 어플리케이션이 있다. Reactive Forwarding App 어플리케이션은 forwarding entry를 연결된 네트워크 스위치들에게 설치하는 것으로 ICMP packet을 보내게 되면 스위치들은 설치된 flow를 보고 패킷을 전송하게 된다. 그림 13과 같이 ICMP packet이 제대로 전송되는 것을 확인 할 수 있다.

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.209 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.072 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.077 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.069 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.078 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.074 ms
```

그림 85 Mininet Ping Test after activation of Reactive Forwarding App in ONOS

9.2.4. Mininet을 이용한 ONOS SDN 제어기 사용 예제: Intent

o ONOS는 기본적으로 Flow를 생성하는 방법에 있어서 Intent subsystem을 제공한다. Intent subsystem은 Flow에 대한 Action, 경로 등에 직접적인 부분에 초점을 두지 않고, 관리자 입장에서 high-level interface를 제공하는 것을 목표로 한다. 이는 어플리케이션들로부터 네트워크 복잡성을 추상화해주는 역할을 한다. ONOS에서 제공하는 Intent의 종류는 “Host to Host Intent”, “Path Intent”, “Point to Point Intent” 등이 있고 이외에도 네트워크 특성에 따른 세세한 Intent가 존재 한다. 그림 14는 ONOS의 Intent Framework를 나타낸다. Intent Framework를 통해 high-level의 Intent를 네트워크 장비들을 세팅하는 device command로 번역해 Flow를 추가하게 된다.

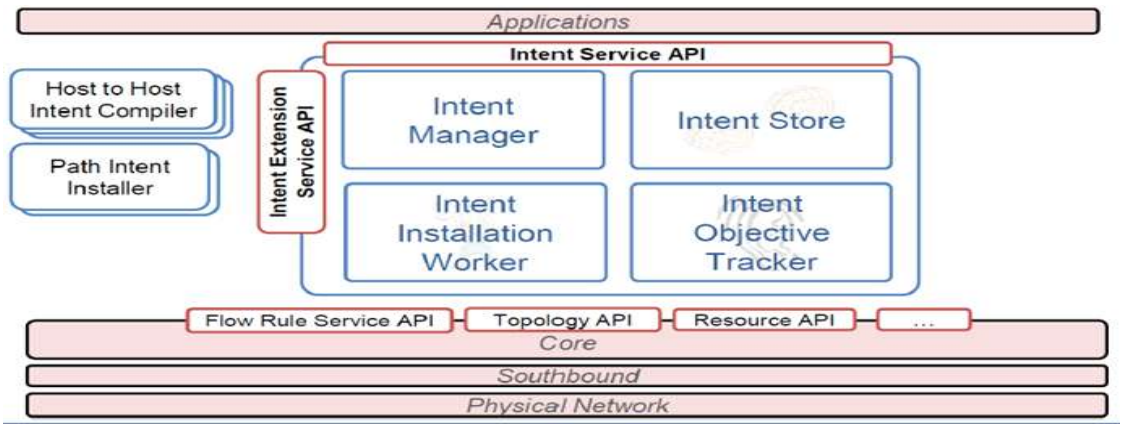


그림 86 ONOS Intent Framework

o 이전에 활성화 시켰던 “Reactive Forwarding App” 어플리케이션을 다시 비활성화 시킨다. 어플리케이션을 활성화 또는 비활성화 시킬 때에는 ONOS 웹 브라우저에서도 가능하지만 CLI를 통해서도 할 수 있다. 그림 15와 같이 ONOS CLI에서 다음과 같은 명령어를 통해 “Reactive Forwarding App” 어플리케이션을 비활성화 시킬 수 있다.

그림 87 ONOC CLI를 통한 어플리케이션 비활성화

o "Reactive Forwarding App" 어플리케이션을 비활성화 한 후, "Host-to-Host Intent"를 통해 원하는 호스트들의 통신이 가능하도록 Intent를 설치한다. Mininet의 가상 호스트들 중, h4 호스트와 h9 호스트의 통신이 가능하도록 "Host-to-Host Intent"를 설치하면 그림 16과 같다.

o Mininet CLI 혹은 ONOS 웹 브라우저의 topology를 보면 h4, h9 호스트들에 대한 각각의 MAC 주소를 확인 할 수 있다. 따라서 그림 16과 같이 "add-host-intent" 명령어를 통해서 h4와 h9의 MAC 주소를 입력하게 되면 h4와 h9에 대한 Intent가 생성되게 되고, CLI와 웹 브라우저를 통해 생성된 Intent를 확인 할 수 있다.

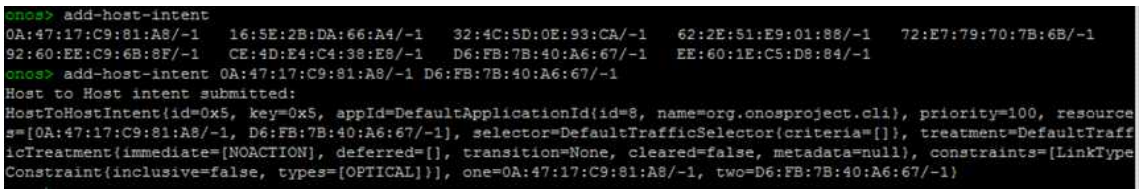


그림 88 ONOS CLI를 통한 Intent 설치

o Intent를 효과적으로 설치 한 후, Mininet에서 ICMP packet을 전송해보면 "Reactvie Forwarding App" 어플리케이션과 같이 정상적으로 전송이 가능하다. 또한 그림 17과 같이 웹 브라우저를 통해서도 설치한 Intent를 확인 할 수 있다.

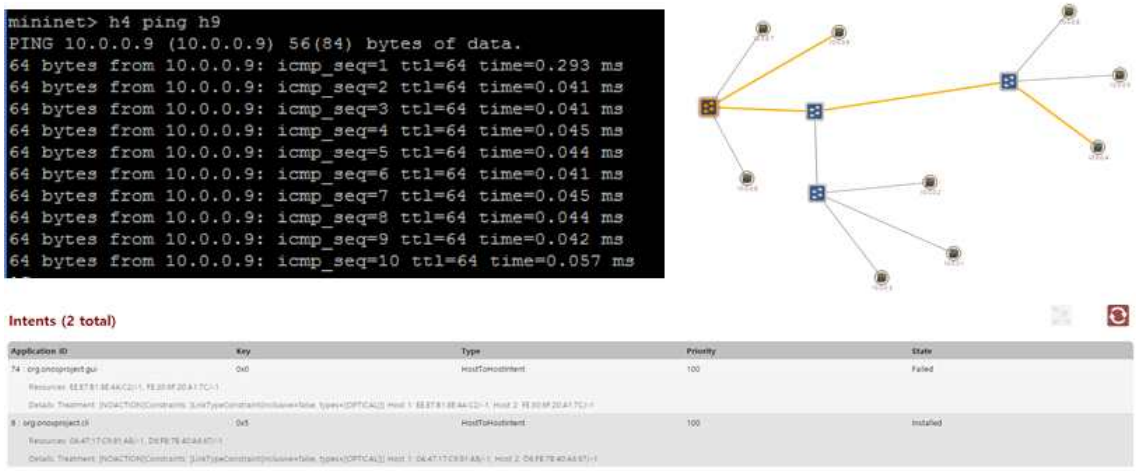


그림 89 ONOS Intent 결과

9.3. 결론

o 본 기술문서 SDN part는 SDN이 생소한, 한번도 경험하지 못한 사람들을 대상으로 작성되었으며 SDN의 개념을 비롯해 OpenFlow, ONOS가 무엇인지, ONOS SDN의 구성 방식과 간단한 사용법을 배워보는 식으로 구성이 된다.

References

[1] http://openvswitch.org/
[2] https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine
[3] https://www.docker.com/what-docker
[4] https://en.wikipedia.org/wiki/Raspberry_Pi
[5] https://technet.microsoft.com/en-us/library/cc776379(v=ws.10).aspx
[6] http://flume.apache.org/
[7] http://kafka.apache.org/documentation.html#introduction
[8] http://kafka.apache.org/documentation.html
[9] https://docs.docker.com/engine/getstarted/
[10] https://hub.docker.com
http://nodejs.org/
http://expressjs.com
http://pyrasis.com/nodejs/nodejs-HOWTO
http://www.codediesel.com/nodejs/processing-file-uploads-in-node-js/
http://www.gliffy.com/publish/2752090/
http://kipid.tistory.com/entry/Learning-Nodejs
http://mesos.apache.org/
http://hadoop.apache.org/
https://spark.apache.org/
http://zeppelin.apache.org/
http://spark.apache.org/docs/latest/api.html
https://wiki.onosproject.org/display/ONOS11/Experimental+Features
https://www.opennetworking.org/about/onf-overview

OF@KOREN 2016 기술 문서

작성기관: 광주과학기술원
작성년월: 2016/11